UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

INGENIERÍA TÉCNICA DE TELECOMUNICACIONES: Sonido e Imagen

PROYECTO FIN DE CARRERA

DESARROLLO DE UN MOTOR GRÁFICO 3D PARA VIDEOJUEGOS DE PLATAFORMAS EN ANDROID.

Autor: Cristina Mingueza Balsalobre
Tutor: Raúl Arrabales Moreno
Director: Jorge Muñoz Fuentes

Agradecimientos

En primer lugar quiero comenzar dando las gracias a mis padres, Mª Ángeles y Urbano. Gracias por vuestro apoyo incondicional, por vuestros consejos, por vuestra ayuda constante, por vuestras palabras de ánimo y por vuestra gran paciencia. Sin vosotros llegar hasta aquí hubiera sido imposible.

A Jorge Muñoz, mi director, por darme la oportunidad y la idea para realizar este proyecto y estar siempre disponible para resolver mis dudas, corregir mis errores y aportarme información y conocimientos.

A mis amigos y compañeros de universidad, en especial al equipo. Sin vuestro apoyo llegar hasta el final hubiera sido mucho más difícil. ¡Somos los mejores!

A Juan Luis, mi compañero de proyecto, gracias por ayudarme día y día y por tus ideas.

Al resto de mi familia, por su confianza, apoyo y cariño.

A Iván, por su apoyo y paciencia, por estar siempre a mi lado en los momentos más difíciles y darme fuerzas para seguir adelante con ilusión.

Resumen

Este proyecto tiene como objetivo principal la implementación de una API [1], para el desarrollo de videojuegos de plataformas en 3D sobre Android. Dicha librería ayudará al programador a crear los escenarios en 3D, los personajes principales, los enemigos y otros elementos necesarios para la construcción de un videojuego de plataformas, como la iluminación de la escena o el uso de la cámara.

El desarrollo del mismo se realizará mediante la herramienta OpenGL ES 1.0, una variante simplificada de la API gráfica OpenGL, diseñada para dispositivos integrados.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ÍNDICE DE CÓDIGO

CAPÍTULO 1.- INTRODUCCIÓN

En el siguiente capítulo se ofrecerá una visión general del alcance del proyecto. Exponiendo las motivaciones que impulsaron su desarrollo, los objetivos perseguidos y los contenidos ofrecidos en esta memoria.

1.1.- Motivación del proyecto

A lo largo de la historia, los principales obstáculos de la comunicación entre seres humanos han sido la distancia y el movimiento. Una de las grandes innovaciones tecnológicas surgió de la necesidad de mejorar las relaciones interpersonales en sectores gubernamentales y empresas privadas, solventando el problema del movimiento. Dicha innovación fue el teléfono móvil. A pesar de ser una tecnología desarrollada e implantada desde el mercado, su uso masivo a dado lugar a notables transformaciones tecnológicas y sociales.

Con el paso del tiempo, el teléfono móvil paso de ser un instrumento básico de comunicación, a transformarse en una herramienta utilizada por la práctica totalidad de los sectores de la población, gracias a la reducción del tamaño de sus componentes y al aumento de sus prestaciones.

Años más tarde, las empresas de telecomunicaciones se plantean un nuevo reto, el de unir dos grandes tecnologías, Internet y los teléfonos móviles. El boom que supuso Internet en su época, junto con el furor de la telefonía móvil, dio lugar a teléfonos móviles más avanzados que no sólo servían para hablar, mandar mensajes cortos y consultar el calendario o la hora, sino también para acceder a Internet, hacer fotos y vídeos y jugar.

El avance de estos teléfonos, dio lugar a la tercera generación y con ella, al desarrollo de los teléfonos móviles más evolucionados, los teléfonos inteligentes o más comúnmente conocidos como, smartphones[31]. Los servicios asociados con la tercera generación, proporcionaron la capacidad de transferir tanto voz, una llamada telefónica, como datos, descarga de programas y juegos, intercambio de email y mensajería instantánea, como ambas cosas simultáneamente, una videollamada.

Todos estos aparatos cuentan con un sistema operativo que consta de varias partes; un kernel[23] o núcleo, que ofrece servicios como la gestión de procesos y el acceso y gestión de memoria, un Middleware[25], que hace posible la existencia de aplicaciones para móviles, un entorno de ejecución de aplicaciones y un interfaz de usuario.

A medida que los distintos modelos de teléfonos móviles van adquiriendo popularidad, los sistemas operativos con los que funcionan van consolidándose en el mercado. Los sistemas operativos más utilizados son: Android, desarrollado por Google; Symbian OS, producto de la alianza de varias empresas de telefonía móvil; iOS, creado por Apple; BlackBerry OS, desarrollado por la empresa canadiense RIM y Windows Phone, creado por Microsoft.

Ya que Android es uno de los sistemas operativos que ha experimentado un mayor crecimiento en el mercado, tenía una mayor preferencia por programar una aplicación o videojuego en este sistema operativo. De esta forma conseguiría familiarizarme con la plataforma, adquirir una gran experiencia y ser capaz de crear mis propias aplicaciones.

Las siguientes gráficas muestran el elevado crecimiento que ha experimentado el sistema operativo Android en un breve espacio de tiempo.

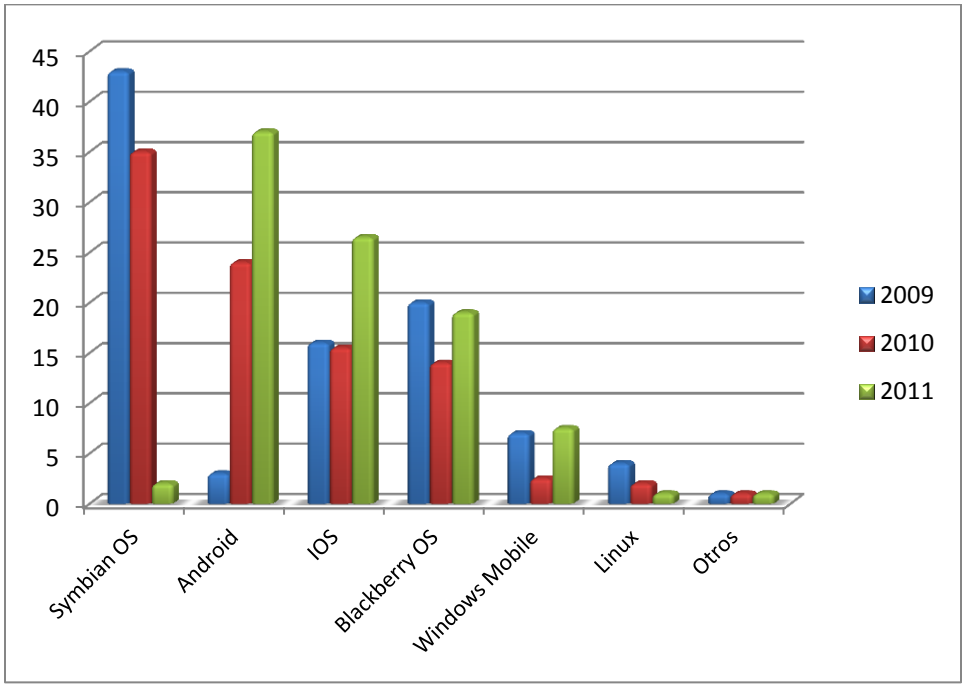


Figura 1: Evolución de la cuota de mercado

1.2.- Objetivos

Los objetivos principales planteados en este proyecto fin de carrera son:

1. La implementación de una biblioteca[3] que permita a un programador crear un videojuego de plataformas en 3D para Android.

2. La creación una memoria explicativa y detallada del proceso seguido para la implementación del proyecto fin de carrera, y los pasos que debe llevar a cabo el programador para hacer usa de la biblioteca implementada.

Para el desarrollo del mismo se utilizaron las siguientes herramientas: la especificación estándar OpenGL ES 1.0, el conjunto de herramientas y aplicaciones de Android, y un teléfono móvil con un sistema operativo Android para probar en él, todos los aspectos técnicos.

Los objetivos personales que se pretenden cubrir son los siguientes:

1. Familiarizarme con los lenguajes de programación y herramientas necesarias para llevar a cabo el proyecto.

   Puesto que el sistema operativo Android se introdujo recientemente en el mundo de las telecomunicaciones, la experiencia previa sobre aspectos como su arquitectura, sus componentes básicos y el manejo de procesos era bastante reducida y fue necesario familiarizarse con ella.

   También fue necesario aprender los aspectos básicos acerca de OpenGL ES, por ser una herramienta fundamental para la implementación del proyecto y no tener conocimientos previos sobre la misma.

2. Aprender técnicas de programación para diseñar sobre el sistema operativo móvil Android, no sólo un motor gráfico en 3D para videojuegos, sino también múltiples aplicaciones.

3. Crear un motor gráfico que permita al usuario implementar un videojuego de plataformas en 3D, de una manera sencilla y libre. Permitiéndole para ello:

    - Configurar un número indefinido de escenarios, compuestos por figuras geométricas. De tal manera que el usuario indicará el lugar y las propiedades de cada una de las figuras que componen la escena, y el sistema se encargará de crear y colocar las mismas.
    - Crear un personaje principal y uno o varios enemigos, cada uno de ellos con unas características determinadas. Dichos personajes se moverán por el escenario 3D, mediante un movimiento fluido y animado.

4. Trabajar en equipo para comprobar la viabilidad de la herramienta realizada, ayudando a otro alumno de la universidad, a construir un videojuego de plataformas en 3D haciendo uso de la misma.

Respecto a los objetivos principales, hay que aclarar que la idea inicial del proyecto era la implementación de un videojuego similar al LittleBigPlanet[24]. Tras un primer análisis se decidió que su desarrollo era demasiado complejo, y por ello se optó por simplificarlo, conservando una característica principal del mismo, el uso del movimiento de la cámara para saber hacia qué lugar del mundo 3D se debería conducir al personaje.

Una vez establecida la idea principal se procedió a separarla en dos partes bien diferenciadas para poder elaborar dos proyectos independientes. Una de estas partes sería la creación de un motor gráfico de videojuegos de plataformas en 3D y la otra probar dicho motor, de tal manera que sólo fuera necesario implementar la lógica del videojuego para construirlo en su totalidad.

1.3.- Estructura del documento

La presente memoria estará compuesta por seis capítulos. A continuación se hará una breve descripción de los aspectos que se profundizarán en cada uno de ellos.

Capítulo 1.- Introducción. Se desarrollan las motivaciones que impulsaron el desarrollo del presente proyecto, una aproximación general de los objetivos principales del mismo, y una visión global de los contenidos de la presente memoria.

Capítulo 2.- Estado del arte. Expone un resumen de la evolución de los videojuegos desde el inicio del primero hasta la actualidad, una breve introducción acerca de los dispositivos móviles y de los distintos sistemas operativos existentes, y el progreso de los videojuegos y de los motores 2D y 3D, en los dispositivos móviles.

Capítulo 3.- Desarrollo. Realiza un análisis de la arquitectura, detallando los casos de uso, los requisitos de usuario y los requisitos software, y mostrando un diagrama de secuencia que explica de manera visual cómo el motor gráfico se encarga de dibujar todos los elementos.

Capítulo 4.- Implementación. Se exponen de manera general las clases que componen la librería creada y se explica detalladamente el modo de construir las figuras, los escenarios y los personajes principales que forman el videojuego.

Capítulo 5.-Pruebas. Expone las pruebas realizadas para el correcto funcionamiento del motor gráfico.

Capítulo 6.- Conclusiones y trabajos futuros desarrollo. Expone las reflexiones obtenidas tras la realización del proyecto, comparándolos con los objetivos marcados inicialmente y traza posibles líneas futuras.

Para finalizar se han incluido una serie de anexos ANEXO A: **Planificación del proyecto**, ANEXO B: **Presupuesto**, ANEXO C: **Resumen detallado del videojuego** realizado a partir del motor gráfico en 3D.

Puntos adicionales: el **Glosario**, que recoge una definición de los términos que pueden resultar ambiguos al lector. Y las **Referencias**, que presenta todas las fuentes que se han consultado para profundizar en los distintos temas expuestos en el proyecto.

CAPÍTULO 2.- ESTADO DEL ARTE

2.1.- El inicio de los videojuegos

A día de hoy es bastante complicado indicar cuál fue el primer videojuego de la historia, a causa de las múltiples definiciones de videojuego que se han ido estableciendo a lo largo del tiempo [1]. A continuación, se explican los juegos que se consideran como los precursores actuales de los sistemas de entretenimiento.

2.1.1.- OXO

En 1952 Alexander S. Douglas, un estudiante de la Universidad de Cambridge, desarrolla en su tesis doctoral titulada, "La interacción entre computadoras y seres humanos", el ***OXO*** [3] una versión electrónica del juego tres en raya, en el cual un ser humano se enfrentaba contra una máquina.

OXO era un juego programado para el ESDAC[10] que utilizaba como control un dial telefónico y como salida, una pantalla de osciloscopio.

Este videojuego se considera uno de los precursores por ser una de las primeras implementaciones con gráficos de salida y una de las primeras demostraciones prácticas de inteligencia artificial.

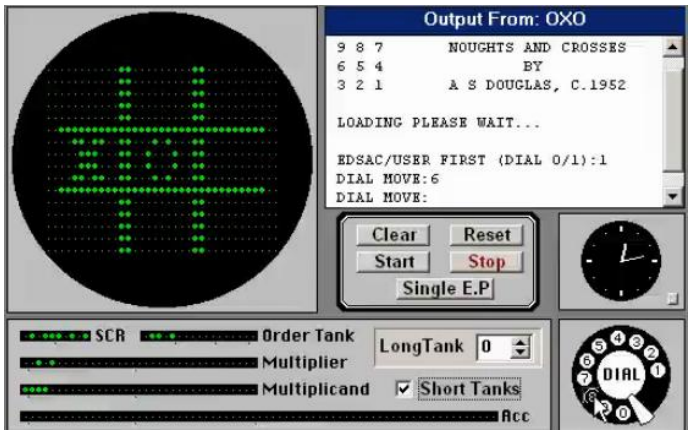


Figura 2: OXO

2.1.2.- Tennis for TWO

William Adolf Higginbotham, era un físico estadounidense que trabajaba como jefe de la división de instrumentación del Laboratorio Nacional de Brookhaven. En 1958, Higginbotham decide diseñar un proyecto con la intención de entretener a los visitantes que acudían a la exposición del laboratorio donde trabajaba. Él, en colaboración con el físico Robert V. Dvorak, crean en tan solo tres semanas el ***Tennis for Two*** [4].

Tennis for Two simulaba un juego de tenis para dos personas que se mostraba sobre un osciloscopio. Dicho juego consistía en dos líneas, una horizontal que representaba la pista de tenis y otra vertical que separaba ambos campos. Para manejar el mismo se hacía uso de dos controladores, uno por cada jugador, formados por un mando analógico que servía para elegir el ángulo con el que salía la pelota, y un pulsador que se usaba para golpearla. A pesar de ser un juego bastante sencillo, permitía al jugador elegir determinados aspectos del mismo, como la altura de la red, la longitud de la pista o el lado del campo de cada jugador.

Tennis for Two sólo fue presentado al público en dos ocasiones como consecuencia de las jornadas de puertas abiertas que tenían lugar en el laboratorio Nacional de Brookhaven. A pesar de que este juego quedó en el olvido, serviría de base para el desarrollo de nuevas aplicaciones.

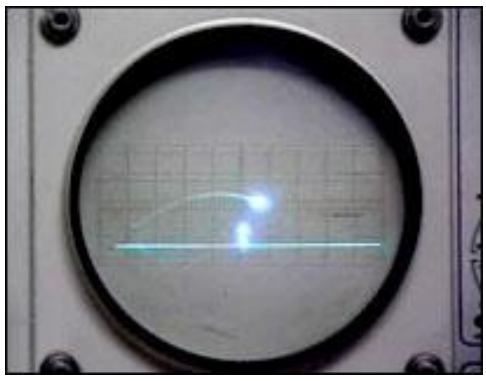


Figura 3: Tennis for TWO

2.1.3. - Spacewar!

En 1961 Steve Russell, Martin Graetz y Wayne Wiitanen comienzan a implementar el juego interactivo ***Spacewar!*** [5] en un computador PDP-1[7] del Instituto Tecnológico de Massachusetts (MIT). La primera versión del juego se termina en febrero de 1962.

Spacewar! es un juego en el que dos naves espaciales provistas de combustibles y misiles limitados, se enfrentan entre sí. El objetivo es destruirse mutuamente sin ser atrapados por la fuerza gravitatoria de una estrella y con la dificultad, de ser atraídos por un agujero negro localizado en alguna de las esquinas de la pantalla.

Progresivamente se fueron añadiendo mejoras al mismo y el juego comenzó a popularizarse. Tal fue su éxito que Digital Equipment, fabricante del PDP-1 [2], incluyó en la fabricación de serie del PDP-1 y versiones posteriores de la computadora, el código del juego.

Es importante mencionar este juego porque aparte de ser el primero en tener cálculos complejos, fue un proyecto que promovió la colaboración e hizo que por primera vez, se incluyese un videojuego dentro del paquete de software que acompañaba a un computador [7].

2.2.- El inicio de las máquinas recreativas

Bill Pitts, graduado de Stanford y su compañero Hugh Tuck, fundaron la empresa Computer Recreations Inc. con la idea de emplazar máquinas en lugares públicos que permitiesen jugar al ***Spacewar!*** a cambio de monedas.

La versión comercial del ***Spacewar!*** diseñada por ambos tomó el nombre de ***Galaxy Game*** [8], pero no tuvo éxito debido a los altos costes de producción de la misma. Sin embargo fue la primera máquina recreativa y por tanto, uno de los grandes avances de la industria de los videojuegos.

Pocos años más tarde, en 1968, Bill Nutting funda Nutting Associates, y consigue solventar los problemas de costes en la producción que había tenido Computer Recreations Inc., gracias a la ayuda de Nolan Bushnell y Ted Dabney.

En 1971, Nutting Associates lanza al mercado el juego ***Computer Space*** [10], un arcade en el que el jugador mediante el uso de los mandos, debía de controlar una nave espacial y matar a todos sus enemigos sin ser destruido en un periodo de 100 segundos.

La máquina tuvo un éxito irregular, pues a pesar de tener una gran aceptación en las universidades no era nada popular en bares y otros establecimientos, por ser un juego bastante complejo [9].

Figura 4: Computer Space

2.3.- La evolución de los videojuegos

2.3.1.- Los inicios de las videoconsolas

En 1966, Ralph Baer consigue llevar a cabo la idea de construir aparatos de ocio para la televisión gracias a la financiación de la empresa donde trabajaba, Sanders Associates. A finales de ese año Baer, junto con su equipo, implementa un dispositivo que permite la aparición de puntos en la pantalla y junto a él, el primer juego, ***Fox and Hounds*** [1]. Poco a poco el proyecto se va desarrollando exitosamente y a finales de 1967, se implementa el juego de ping pong y el primer prototipo de consola conocido como ***"Brown Box"***.

Mientras tanto, Sanders Associates tenía la labor de encontrar a un fabricante dispuesto a correr el riesgo de invertir en la revolucionaria idea de Baer. Pese a la dificultad de dicha labor, Sanders Associates vende todos los derechos de comercialización al fabricante Magnavox [12].

En 1972 ***"Brown Box"*** es lanzada al mercado norteamericano por la filial Philips con el nombre de Magnavox Odyssey. El producto se convirtió en un éxito de ventas en poco tiempo a pesar del elevado precio del mismo. En solo un año, se vendieron aproximadamente 100.000 unidades [11].

Tal fue el éxito que la empresa que la vendía extendió el rumor de que la consola sólo funcionaba en televisores de la misma marca que la consola, Magnavox, para subir al mismo tiempo el volumen de ventas de sus televisores.

La ***Magnavox Odyssey*** [11] era un aparato compuesto exclusivamente por transistores, resistencias y condensadores. Al estar formada por componentes analógicos, la hacía una máquina bastante limitada, pues carecía de sonido y de capacidad de almacenamiento, era incapaz de guardar partidas ni retener ningún tipo de información. Los cartuchos de los juegos no tenían componentes internos, sólo eran unas tarjetas que provocaban conexiones entre los diferentes pines del slot donde se introducían consiguiendo así, generar diferentes señales analógicas que se trasmitían al televisor.

Los gráficos eran bastante limitados, sólo era capaz de dibujar puntos móviles y barras verticales y horizontales en la pantalla. Por ello, la consola se vendía junto con un set adicional que incluía unas plantillas transparentes con dibujos de colores que se pegaban a la pantalla de la televisión para simular gráficos complejos. También se incluían dos mandos, seis tarjetas de juegos, billetes, cartas, una ruleta, fichas de póker, dados y una pizarra para anotar las puntuaciones de los jugadores [12].

En principio la consola solo incluía seis juegos como se ha mencionado anteriormente, entre los cuales se encuentran el tenis, el Simón dice o el ping pong. Pero con el paso del tiempo esta lista se incrementó llegando a incluir hasta doce tarjetas de juegos.

El 1 de Junio de 1972 Nolan Bushnell y Ted Dadney, deciden formar una empresa de pequeño ámbito, llamada ***Atari*** [16]. Unos meses después, Allan Alcorn se incorpora a la misma con el fin de desarrollar un juego similar al ping pong, llamado ***Pong***. En septiembre de ese año, el proyecto está finalizado e instalado en una máquina recreativa. Debido a la gran aceptación por parte del público, Atari se ve obligado a ampliar su negocio, llegando a fabricar un total de 100 máquinas diarias.

Pong [16] está considerado por muchos como el más importante de entre la primera generación de videojuegos modernos, debido a que fue el primero en comercializarse a nivel masivo y no ejecutarse en máquinas únicas.

Al poco tiempo y de forma paralela, empezaron a aparecer los primeros clones de ***Pong*** en los salones recreativos y junto con ellos los temores de Atari crecieron. A causa de ellos, Ted Dadney decide vender su parte de la empresa a Bushnell y Bushnell decide aliarse con Joe Keenan para formar Kee Games, y así hacerse con prácticamente la totalidad del mercado.

Durante los siguientes años ambas empresas intentaban aparecer en el mercado como dos compañías diferentes. Atari lanzaba al mercado máquinas recreativas como ***Space Race***, ***Pong Doubles***, similar al ***Pong*** pero para cuatro jugadores, o ***Gotcha***, precursor de los juegos de laberinto como ***Pac-Man***. Y Kee Games juegos como ***Tank***.

Tank era el primer juego que conseguía almacenar los gráficos en chips de memoria ROM, esto le permitía mostrar en la pantalla detalles gráficos mucho más complejos. El juego consistía en dos tanques que se enfrentan en un laberinto mientras intentan evitar las minas esparcidas sobre el terreno del mismo.

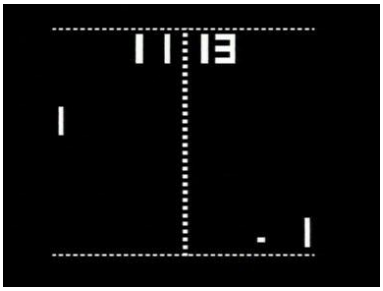


Figura 5: Pong

Figura 6: Tank

Tal fue el éxito de ***Tank***, que Kee Games se empezó a convertirse en un competidor peligroso hasta para Atari. Por esa razón y unido a las pérdidas que le estaban ocasionando a Atari el diseño de un juego denominado "***Grantrak 10***", Bushnell decide dar a la luz la verdad acerca de Kee Games y Atari. A causa de esto, Joe Keenan pasa a ser el presidente de Atari.

El año 1973 fue destacado por el comienzo de una etapa, la del desarrollo de los motores gráficos 3D, que durará hasta aproximadamente el año 1990. Esta etapa se caracterizó por el surgimiento de los primeros entornos pseudo-3D. Los escenarios de los videojuegos eran bastante sencillos y su implementación era muy costosa en relación con el hardware de la época [26]. Dos de los juegos más destacados que se explicarán a lo largo de este capítulo son, ***Maze War*** (1973) y ***Battlezone*** (1980).

En ese mismo año Steve Colley comienza a desarrollar el primer videojuego 3D de disparos en primera persona o FPS[11] en la computadora Imlac PDS-1. El videojuego recibió el nombre de ***Maze War*** [28] y no se finalizó su implementación hasta un año más tarde.

La Imlac PDS-1, al ser una computadora central, permitía jugar hasta 32 jugadores de manera simultánea. El objetivo de los mismos era moverse a través de un laberinto y matar a sus adversarios, el resto de los jugadores.

Figura 7: Maze War

Hasta entonces, Atari [16] se había limitado a trabajar en proyectos de máquinas recreativas, pero era el momento de empezar a llevar sus juegos a los hogares. En 1974 se construye Darlene, un prototipo de la versión domestica de Pong. Y "Home Pong" sale al mercado bajo el nombre de "Pong Telegames by SEARS". Pero Atari tenía que seguir incrementado sus ganancias así que, poco tiempo después lanza al mercado "Super Pong", su segunda consola.

Figura 8: Pong Telegames by SEARS

Figura 9: Super Pong by Atari

Como era de esperar, ante las ganancias que ocasionaban el negocio de las videoconsolas, comenzaron a aparecer competidores. En 1975 Coleco [54] entra en el mercado con Telstar, una de las consolas más deseadas por los consumidores. A pesar de su gran comienzo la empresa tuvo grandes pérdidas por un problema en la producción.

En ese mismo año la empresa Taito saca al mercado ***Gunfight*** [52] un juego para dos, que trata de simular los duelos típicos de las películas del oeste. Midway rediseña la versión original y usa por primera vez en la historia un microprocesador en una máquina arcade.

En 1976 Atari [35] comienza a trabajar en un proyecto denominado Stella, pero debido a un error de cálculo de presupuesto, Nolan Bushell se ve obligado a vender su empresa a la compañía Warner Communications. Dos años más tarde Nolan Bushell es despedido. El proyecto finaliza con la creación de uno de los primeros sistemas de videojuegos domésticos de juegos intercambiables: Atari VCS, que en 1982 cambiaría su nombre por el de Atari 2600.

En ese mismo año Atari saca al mercado dos juegos destacados:

- El primero de ellos es ***Night Driver*** [55], un juego de conducción, en el que se da una perspectiva innovadora para el usuario, la perspectiva en primera persona. El juego se desarrolla en un ambiente nocturno, de esta forma se oculta al usuario carencias de la tecnología para crear imágenes más complejas. Años más tarde se diseñaron juegos en 3D construidos a partir del concepto básico de ***Night Driver***, usando hardware computacional para simular el movimiento 3d mediante imágenes planas llamadas sprites[32].

Figura 10: Gunfight

Figura 11: Night Driver

- El segundo juego fue diseñado por Steve Jobs y Steve Wozniak para Atari, justo antes de que formaran la compañía de computadoras Apple, y recibe el nombre de ***Breakout*** [52]. El objetivo de ***Breakout*** era romper una pared de ladrillos situada en la parte superior de la pantalla gracias a una pelota que el usuario golpea de arriba abajo con la ayuda de una raqueta situada en la parte inferior. Años más tarde saldrían al mercado innumerables variaciones del mismo para todo tipo de consolas y soportes, entre ellas, Arkanoid.

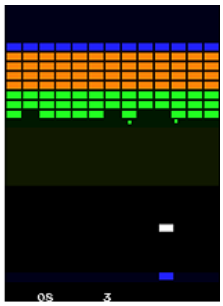


Figura 12: Arkanoid

En 1977 [14] Yamauchi decide introducir su empresa, Nintendo, en el mercado de los videojuegos. Para ello crea su propia versión de la consola Pong llamada Color TV Game 6, que sólo fue comercializada en Japón. La consola fue un éxito rotundo para Nintendo, tanto que sacaron 4 versiones más de la misma, entre ellas, Color TV Game 15.

Figura 13: Color TV Game 6

Nintendo no solo comenzó a fabricar consolas sino también videojuegos. En 1978 desarrolla el primer juego para máquinas recreativas denominado ***Othello*** [29], un arcade basado en el juego de mesa ***Othello***. A pesar de no tener demasiado éxito, es un dato importante a desatacar en la historia de Nintendo, pues la empresa comenzó con el diseño de videojuegos que pasarían a la historia.

En 1978 Toshihiro Nishikado, uno de los programadores de Taito Corporation, diseña el ***Space Invaders*** [30], un juego que revolucionó el mercado nipón. El juego tiene como objetivo principal la destrucción de cinco filas de marcianos que se van desplazando de un lado a otro de la pantalla y que cada vez se aproximan más rápidamente hacia la posición de la nave.

Tal fue el éxito de ***Space Invaders,*** que no sólo revolucionó el mercado japonés, sino también el americano. Midway Games, empresa estadounidense desarrolladora de videojuegos, se encargó de la distribución del mismo en América.

En 1978 salen al mercado nuevas consolas como la Magnavoz Odyssey 2 y una de las mejoras de Nintendo la Nintendo TV Game 15, anteriormente mencionada.

Un año más tarde Atari saca al mercado ***Asteroids***, un juego que acabaría desplazando al mítico, Space Invaders y que le convertiría en la empresa de videojuegos más exitosa del mercado. El objetivo del juego es destruir mediante el botón de disparo cuatro asteroides que se mueven de forma aleatoria por la pantalla y que una vez disparados, se dividen una y otra vez en trozos más pequeños. El juego finaliza en caso de que algún asteroide golpe la nave, o que la nave, acabe con todos los asteroides de la pantalla.

Asteroids [53], no sólo se caracterizó por su éxito, sino por introducir mejoras en los videojuegos que incluso a día de hoy se siguen usando, como el sistema de refresco de vectores denominado Quadrascan, o el registro de las iniciales de los jugadores para las marcas de puntuación.

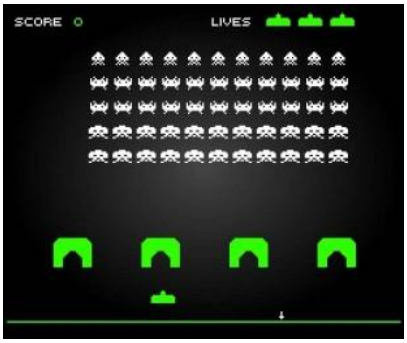


Figura 14: Space Invaders

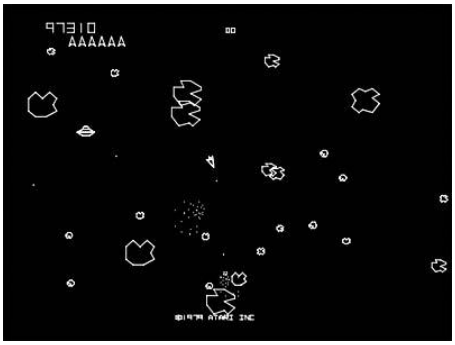


Figura 15: Asteroids

2.3.2.- La década de los 8 bits

Los años 80 comienzan con un gran crecimiento en la industria de los videojuegos en parte producido por la popularidad de las máquinas recreativas y de las primeras videoconsolas.

El año 1980 se recuerda como uno de los años más relevantes de este periodo, con productos tan destacados como:

- El videojuego arcade ***Pac-Man*** [56] lanzado al mercado por la empresa Namco. Este videojuego fue diseñado por Toru Iwatani, programado por Hideyuki Mokajima, y distribuido por Midway Games al mercado estadounidense.

Pac-Man, se convirtió en un fenómeno mundial en la industria de los videojuegos. Tal fue su éxito, que llego a tener el Record Guiness del videojuego de arcade más exitoso de todos los tiempos. Incluso a día de hoy se sigue utilizando.

Se cree que la gran aceptación del mismo fue debido a su innovadora idea, en lugar de ser un juego de disparos, como la mayoría de los juegos hasta el momento, se cambia la temática del mismo por la de un muñeco amarillo que tiene como meta, comerse todos los puntos de un laberinto mientras es perseguido por cuatro fantasmas de distintos colores.

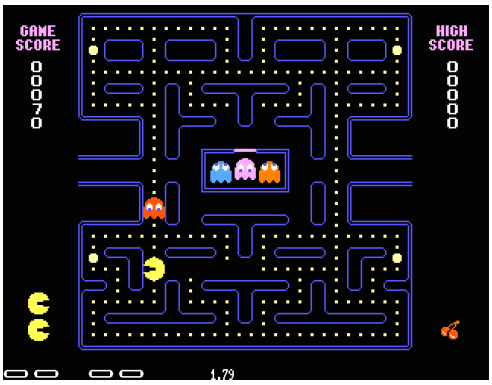


Figura 16: Pac-Man

- Los **Game&Watch** [20], un juguete electrónico similar a una videoconsola portátil, que constaba de una pantalla LCD como las de los relojes digitales o las calculadoras y que fue puesta en el mercado por Nintendo.

Figura 17: Game&Watch

- **Mattel Intellivision** [37] una de las mejores consolas de la primera generación, de Mattel, que durante años compitió en el mercado con Atari 2600. A pesar de tener unos gráficos y unos mandos superiores, no tuvo demasiado éxito.

Mientras, en aquella época, Shigeru Miyamoto, diseñador y productor de videojuegos de Nintendo, trataba de buscar un personaje para la edición de un videojuego revolucionario. Dos de sus personajes favoritos eran King Kong y Popeye, por ello, intentó hacer un videojuego que tratase de una historia entre ambos, pero por problemas de derechos le fue imposible utilizar la imagen de ambos personajes. A causa de ello, se vio obligado a crear otros nuevos entre ellos Jumpman que posteriormente pasaría a la historia con el nombre de Mario y Donkey Kong, copia del famoso King Kong que tanto le apasionaba [40].

En el año 1981, sale el juego de Miyamoto con el nombre de ***Donkey Kong*** [18]. En el juego, Jumpman, debía rescatar a una princesa de las garras del gorila, Donkey Kong. Tales fueron las ganancias de Nintendo por el éxito del mismo, que Miyamoto se vio en la necesidad de profundizar en la figura de Jumpman, el protagonista**.**

Battlezone [41] de Atari, fue considerado el primer videojuego arcade de realidad virtual. Este contaba con un impresionante diseño futurista en el que el jugador utilizaba un visor que le permitía introducirse en un escenario vectorial 3D en primera persona. El jugador debía manejar un taque y su objetivo era destruir los tanques enemigos y otros objetos no identificados. Este juego impresionó tanto a las Fuerzas Armadas estadounidenses que encargaron a Atari una versión mejorada del mismo para el entrenamiento de sus pilotos de tanques.

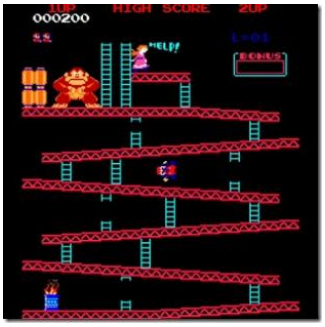


Figura 18: Donkey Kong

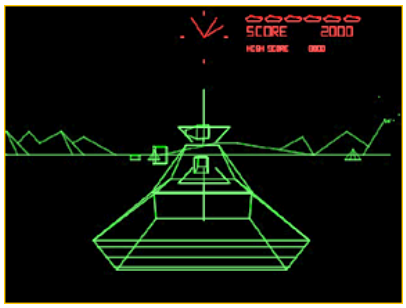


Figura 19: Battlezone

En 1982 los mejores trabajadores de Atari comienzan a abandonar la empresa por la mala política de Warner Communications. Unido a esto, la empresa comienza a generar pérdidas pues decide sacar al mercado dos juegos que no tienen éxito alguno; una pésima versión de ***Pac-Man*** y el juego de ***ET el extraterrestre***, por el que pagan una suma importante de dinero a los estudios por la compra de la licencia [35].

En ese mismo año, cuatro de los mejores programadores de Atari se incorporan a la plantilla de Activision y diseñan ***Pitfall*** [38], el primer juego de plataformas de la historia, que alcanzó un total de cuatro millones de copias vendidas. El objetivo del juego era conducir al personaje principal, a través de una serie de pantallas ambientadas en la selva sorteando por medio de saltos, obstáculos como animales o charcas y consiguiendo tesoros ocultos.

Figura 20: Pitfall

En este año también aparecen consolas de segunda generación, entre ellas podemos encontrar las siguientes:

- **Arcadia 2001,** una consola de 8-bits publicada por Emerson Radio Corp. El objetivo de Emerson era crear una consola de pequeño tamaño y mucho más potente que la consola líder del mercado, la ya conocida Atari 2600. Sin embargo el objetivo de Emerson no se cumple, pues poco tiempo después aparece en el mercado Colecovision, una consola de mejores características.

- **Colecovision** [37] es lanzada al mercado de Estados Unidos por Coleco. Esta consola, era técnicamente superior a todas las de la competencia, no solo por emular bastante bien las máquinas recreativas sino también por ser el primer sistema en incorporar un adaptador para jugar con los juegos de la competencia. Eric Bromley, director de diseño y desarrollo de juegos de Coleco, decide vender junto a la consola un juego revolucionario. Por ello, pacta con Nintendo para vender junto a la misma el juego Donkey Kong.

- **Atari 5200** [39] la nueva consola de Atari. Esta consola no tuvo tanto éxito como Atari 2600 pero era una máquina muy potente, que ofrecía gráficos bastante nuevos para la época.

- **ZX Spectrum** [20] de Sinclair. Esta consola poseía un procesador de 8 bits a 3,58 Mhz con 16KB de memoria. Los juegos de la misma venían en cintas de casete. Debido a la falta de recursos de la misma pasa a convertirse en el ordenador más pequeño, barato y popular de la década de los 80 de todo el mercado europeo.

- **Vectrex**, fue la primera consola en basar sus juegos en vectores, este hecho le daba cierta ventaja sobre sus competidoras al ser capaz de recrear entornos 3D.

El año 1983 es un año muy crítico para Atari pero sin embargo el mercado de los juegos y videoconsolas vivía su mejor momento.

En este año salen los primeros videojuegos relacionados con el mundo del cine, algunos ejemplos de los mismos son ***Tron***, basado en una la película de Disney, ***Star Wars*** de Atari o la ***segunda parte de Tron***.

También aparecen en el mercado dos juegos de Nintendo, el ***Donkey Kong III*** y el famoso ***Mario Bros***. ***Mario Bros*** [1] es el primer juego de plataformas de desplazamiento lateral diseñado para máquinas recreativas de Nintendo. El objetivo del mismo es conducir a Mario, anteriormente conocido como Jumpman, o a Luigi, su hermano, a través de distintos niveles, en ellos tendrán que derrotar a sus enemigos. La dificultad del juego aumentará a medida que el personaje pase de nivel.

Figura 21: Mario Bros

Más tarde Nintendo lanza en el mercado japonés su primera videoconsola de ocho bits independiente, la Famicom [57]. Que llegaría a Europa y Estados Unidos un par de años más tarde con el nombre de N.E.S (Nintendo Entertainment System). Durante los primeros meses Famicom consiguió un gran número de ventas, hasta que se detectó un fallo técnico en el montaje de la misma, y cesó temporalmente la distribución. Una vez solventado el problema, la popularidad se disparó.

Pero Nintendo no es la única empresa que lanza una videoconsola en este año, Sega lanza la SG-1000**.**

Es a finales de este año cuando comienza la llamada crisis del videojuego, que afecta principalmente a países como Estados Unidos y Canadá y que no terminaría hasta 1985.

En 1984 a causa de la grave crisis que sufría el mercado de los videojuegos algunas compañías como Universal y la sucursal norteamericana de Sega se hundieron [20]. Sin embargo, otras empresas como Capsule Company, más conocida con el nombre de Capcom, aparecieron.

En 1985 la industria de los videojuegos comenzó a recuperarse. Algunos de los juegos más destacados en este año fueron ***Super Mario Bros*** y ***Tetris***.

Super Mario Bros [58] es un videojuego de plataformas, diseñado por Shigeru Miyamoto, y producido por la compañía Nintendo, para la consola Nintendo Entertainment System. El objetivo del mismo es conducir a Mario o a Luigi hacia el final ocho niveles repletos de tuberías y de monstruos, con el fin de rescatar a la princesa Peach de las garras de su enemigo, Browser.

El estreno de ***Super Mario Bros*** supuso un punto de inflexión en el desarrollo de los juegos electrónicos, pues sería el primer juego con una temática y un objetivo distinto al de todos los juegos diseñados hasta esa fecha. Este juego es considerado el juego más vendido de todos los tiempos, tal fue su éxito que Mario se convirtió en la mascota principal de la compañía.

Tetris [59] es un videojuego de puzzle inventado por el ruso Alexey Pajitnov. Alexey Pajitnov era un investigador de inteligencia artificial que trabajaba en la Academia de Ciencias de Moscú. Durante su estancia allí, recibió un nuevo ordenador denominado Elektronika 60 y decidió probarlo programando un juego simple y divertido. Así nació en una oficina de investigación de la Unión Soviética, ***Tetris***, un juego diseñado a partir de la idea del juego de pentaminós[28].

Al poco tiempo ***Tetris*** comenzó a ganar popularidad gracias a la adaptación del juego que hicieron Dmitry Pavlovsky y Vadim Gerasimov, dos compañeros de trabajo de Alexey Pajitnov, para el ordenador IBM PC[20]. Atari y Nintendo lucharon por su licencia consiguiéndola finalmente Nintendo gracias a Henk Rogers.

Tetris ha sido uno de los juegos más versionados. El objetivo del mismo es colocar ordenadamente en la parte inferior de la pantalla una serie de figuras geométricas que caen de la parte superior, para formar una línea horizontal. Cuando la línea es completada esta desaparece dejando espacio para seguir colocando piezas. El juego acaba cuando las piezas se amontonan hasta salir del área de juego.

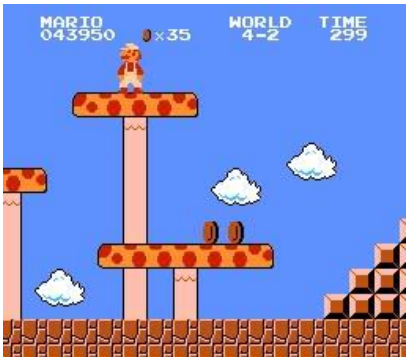


Figura 22: Super Mario Bros

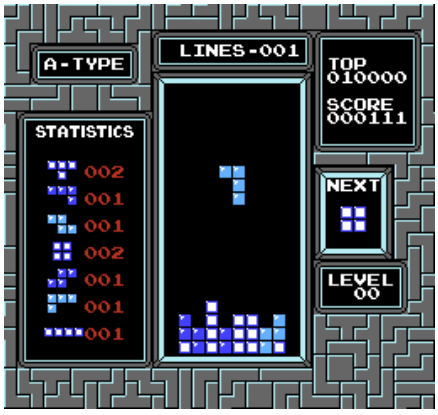


Figura 23: Tetris

En 1986 [1] se lanzaron al mercado una gran cantidad de videojuegos. En el campo de las recreativas se pueden destacar juegos como ***Out Run*** de Sega y ***Arkanoid*** y ***Bubble Bobble*** de Taito. A continuación se procederá a explicar Out Run uno de los juegos más novedosos de ese año.

Out Run [60] es un arcade de conducción creado por Yu Suzuki para Sega. Esta máquina recreativa se caracteriza por ser la primera en introducir elementos electro-neumáticos que la hacían moverse de tal forma que simularan los movimientos del coche cuando este circulaba. Además de esto, ***Out Run*** se conoce por ser uno de los primeros juegos en recrear técnicas como el Sprite Scaling y el Super Scaling. Estas técnicas dotaban al juego de unos gráficos impresionantes y realistas, que no serían superados hasta un año más tarde con la aparición de los juegos de carreras poligonales. En él, se conseguía que no sólo aumentara el tamaño de los oponentes sino que también se modificaran en tiempo real los elementos que decoraban el escenario a medida que el jugador se acercaba a ellos. Otro de los aspectos que catapultó a ***Out Run*** fue su genial banda sonora.

Figura 24: Out Run

También nacen juegos como el ***Legenda of Zelda***, ***Castlevania*** y ***Metroid***, diseñados para la consola Famicom de Nintendo [20].

En este año también salen al mercado varias consolas. Sega lanza la SG-1000 Mark III en Japón, que un año más tarde sería rediseñada para venderse en el mercado estadounidense bajo el nombre Master System. Esta consola tenía unas características técnicas superiores a la NES de Nintendo, sin embargo no alcanza la misma popularidad. Nintendo sigue teniendo la mayor parte del mercado a pesar de los esfuerzos de Sega. Atari ante el éxito de Nintendo y Sega decide competir con ellos sacando al mercado la 7800

En 1987 [20] Sega, Capcom y Taito se hacen con la totalidad del mercado gracias a la calidad de sus lanzamientos. Los videojuegos más desatacados de Sega fueron ***After Burner***, ***Alien Syndrome***, ***Shinobi***, ***Heavyweight Champ***, ***SDI -Strategic Defense Initiative*** y ***Wonder Boy In Monster Land***. Entre los juegos de Capcom, solamente dos son los más importantes ***Battle Of Midway*** y ***Street Fighter***.

Consolas como la NEC PC Engine de NEC y el Amiga 500 y 2000 de Commodore son diseñadas y lanzadas al mercado.

- La NEC PC Engine era un sistema de 8 bits con un chip gráfico de 16 bits y con una velocidad conmutable entre 3,58 Mhz y 7,6 Mhz que llego a tener muchísima aceptación en Japón, tanta que incluso llego a vender más que la Famicom durante los primeros meses de su lanzamiento.

- La Amiga 500 tenía un procesador a 7,16 Mhz con 512k de RAM y 256k de ROM. Era el primer ordenador personal que permitía correr varios procesos a la vez, por ello tuvo un gran éxito entre los consumidores.

- La Amiga 2000 estaba más orientada al entorno profesional ya que en lugar de 512k de RAM tenía 1Mb, pero era muy similar a la Amiga 500.

Figura 25: NEC PC Engine

Figura 26: Amiga 2000

En 1988 Sega, ante el fracaso de su anterior consola, decide lanzar al mercado la Mega Drive en Japón, una consola de 16 bits. Poco a poco, gracias a su avanzado hardware, Sega consiguió tener mayor popularidad que Nintendo hasta la aparición de la Super Nintendo.

En el 1989 se lanzan las primeras videoconsolas portátiles, Atari Lynx y la Game Boy.

2.3.3.- La revolución del 3D

La década de los 90 es considerada como la época dorada de los videojuegos. Durante estos años, se desarrollan un gran número de motores gráficos que permiten la creación de videojuegos con entornos en 3D realistas.

En 1990 llega al mercado la Super Famicom [21], también conocida como la Super NES, de Nintendo. La consola era muy superior a la de todos sus competidores, ya que tenía un procesador de 16 bits con velocidades variables de 1.79, 2.68 o 3,58 Mhz y se podían jugar a numerosos juegos que fueron saliendo durante los años de vida de la misma. Debido a superioridad, en poco tiempo se convirtió en una gran rival para la Mega Drive de Sega.

Pero Nintendo no es la única que saca nuevas consolas, Sega lanza la Game Gear [17]una alternativa a la Game Boy de Nintendo y la Master System II, una nueva consola mucho más ligera y pequeña. Además SNK pone a la venta Neo Geo, una consola con una alta tecnología y de elevado coste que destacaba por el gran colorido y el fenomenal sonido de sus videojuegos. También se diseñan juegos como el ***Super Mario World*** de Nintendo, que presentaba mejoras en los gráficos y en los sonidos. Este juego es un completo éxito en todo el mundo, tanto que a día de hoy es considerado uno de los mejores juegos de la historia.

Este año también es destacado por el inicio de la segunda etapa de la evolución de los motores 3D [26], que no terminará hasta 1995. En esta etapa, los mundos en pseudo-3D son mucho más elaborados, además se comienzan a usar de manera masiva los sprites y aparecen las texturas[34] planas. También hay un aumento bastante considerable en la cantidad de polígonos y se comienzan a utilizar la iluminación y las sombras. Los juegos que marcarán esta época y que se explicarán a lo largo de este capítulo son: **Catacomb 3D**, **Wolfenstein 3D**, **Alone in the Dark**, **Doom**, **Doom2** y **Duke Nukem 3D**.

En 1991 Sega lanza ***Sonic*** [21], un juego que pretendía hacer frente al famoso Mario de Nintendo. El juego se convirtió en un gran éxito mundial tanto que llegaría a los 4 millones de copias vendidas.

También se lanza ***Catacomb 3D*** [42] un juego de disparos en primera persona para la plataforma IBM PC. El juego introdujo la idea de mostrar la mano del jugador en la vista 3D. En él, manejamos a Petton Everhail, un mago que debe aventurarse en las oscuras catacumbas para poder rescatar a su amigo de las garras de su enemigo Némesis. El juego utilizaba gráficos EGA[18], por ello, tan solo podía reproducir 16 colores.

Figura 27: Sonic

Figura 28: Catacomb 3D

En este año se pone punto y final a la *Edad de Oro del Software Español.* Tras la llegada en los años 90 de los 16 bits al mercado europeo, muchas desarrolladoras españolas fueron desapareciendo, pues se quedaron estancadas en la época de los 8 bits.

En el año 1992 aparecen juegos exitosos como la segunda parte de ***Sonic*** o el ***Mortal Kombat***. Sin embargo, los juegos que se detallarán a continuación debido al gran avance que ocasionaron en el mundo del 3D serán, ***Wolfenstein 3D*** y ***Alone in the dark***.

- ***Wolfenstein 3D*** [64] se caracteriza por ser el primer juego en aprovechar la perspectiva tridimensional. El motor gráfico era bastante escaso puesto que no tenía ni suelos, ni escaleras, ni techos y los objetos eran sprites perpendiculares al jugador, pero a pesar de esto, fue totalmente revolucionario y supuso un salto de calidad de los ordenadores sobre las consolas.

- ***Alone in the Dark*** [25] es un juego del género Survival Horror[33] y fue uno de los primeros que utilizó un entorno 3D tal y como lo conocemos en la actualidad. Gráficamente era una maravilla.

  El juego se desarrolla en una mansión cuyo dueño se ha suicidado y resulta estar repleta de zombies y criaturas horrorosas. Al inicio del juego se permite al jugador elegir entre dos personajes, uno de ellos es Edward Camby, un detective cuya misión es realizar un inventario de los objetos de la mansión y el otro es Emily Hartwood, la sobrina del dueño, que tratará de averiguar el porqué de la muerte de su tío. El objetivo común de ambos es sobrevivir y huir de la mansión.

Figura 29: Wolfenstein 3D

Figura 30: Alone in the Dark

En 1993 sale al mercado una mejora de la Mega Drive, la Mega Drive II. También salen juegos como el ***Doom*** y el primer ***FIFA*** de todos.

Tras el éxito de ***Wolfenstein 3D***, Id Software comienza a desarrollar ***Doom*** [61], un juego 3D en primera persona. El juego tuvo una gran aceptación entre el público y durante mucho tiempo fue líder de ventas entre los juegos de PC, pues a pesar de lo limitado que era el hardware de la época, la atmósfera conseguida, el aspecto de los enemigos, las armas disponibles y la música de fondo, eran inmejorables.

Para el desarrollo del mismo y de su predecesor, Doom II, Id Software usó el motor gráfico ***Doom Engine*** [22], creado por John Carmack junto con la ayuda de Mike Abrash, John Romero, Dave Taylor y Paul Radek. ***Doom Engine*** no es un motor 3D propiamente dicho, ya que si observamos un nivel de Doom desde una perspectiva área se puede apreciar que es de 2D, es decir, no se puede tener unas salas sobre otras. Debido a esto, ***Doom Engine*** es considerado un sistema que permite renderizado[30] en pseudo-3D. Este sistema fue revolucionario gracias a su habilidad para proveer un ambiente 3D rápido por mapeado de texturas[26].

Este motor consta de unos objetos básicos: los vértices, denominados *vertex* que representan un punto en 2D, las líneas llamadas *linedefs* que son formadas a partir de la unión de dos vértices y los lados, *sidedefs* cuya unión formarán áreas particulares del nivel, los polígonos denominados *sectors*. A la hora de editar un polígono, se deberán de establecer unas propiedades al mismo como la altura, el nivel de iluminación o la textura. A parte de estos elementos hay una lista de objetos en cada nivel. Estos objetos, son conocidos como *things* y sirven para representar al jugador o a los monstruos en una coordenada 2D específica. La estructura de datos que se utilizó para organizar los objetos dentro del espacio fue BSP[4] (Particionado binario del espacio).

Pocos meses más tarde sale ***FIFA*** [21] al mercado, un videojuego deportivo diseñado por EA Games. Este juego proporcionaba una vista isométrica[21], algo novedoso para aquella época. También tenía una gran cantidad de animaciones, un novedoso sistema de repeticiones y una buena inteligencia artificial.

En 1994 llega a Japón la primera consola de sobremesa de Sony, la PlayStation. Esta consola, se caracteriza por ser la primera en emplear el CD-ROM como soporte de almacenamiento para sus juegos con una velocidad de 2x y por tener una CPU de 32 bits a 33,8 Mhz. Gracias a estas características, se convierte en todo un éxito tanto que llega a ser una de las más vendidas de la historia.

SNK lanza la Neo Geo CD. Puesto que su última consola no llego a tener demasiado éxito debido al coste de la misma, decidieron sacar otra, esta vez accesible económicamente a todo el mundo. Esta consola de 16 bits tenía el mismo hardware que su antecesora pero incorporaba una RAM superior, también tenía una nueva unidad de CD con una velocidad 1x, una velocidad bastante baja, que hacía que los juegos tardarán demasiado tiempo en cargarse. Este pequeño inconveniente, hizo que SNK lanzara más adelante una consola que solventara este problema, la Neo Geo CDZ.

Figura 31: Neo Geo CD de SNK

Figura 32: Sony Playstation

Este año también se conoce por el desarrollo de videojuegos de disparos en primera persona para computadora como ***Doom II: Hell on Earth*** de Id Software y ***Heretic*** y ***Hexen*** de Raven Software [25].

- ***Doom II*** marcó una antes y un después en la historia de los videojuegos en 3D, incluso llegó a convertirse en el mejor de su género. El juego se desarrolla en la Tierra y el objetivo del mismo es acabar con unas criaturas del infierno que han entrado en la Tierra a través de un portal interdimensional.

- ***Heretic*** era un videojuego en 3D que usaba un motor gráfico bastante similar al de Doom. Las armas, los objetos y algunos monstruos en ***Heretic*** tenían su equivalente en Doom. Este detalle pasó desapercibido gracias al diseño gráfico del juego.

Figura 33:Doom II

Figura 34: Heretic

Dos años más tarde en 1996, Nintendo lanza dos nuevos productos al mercado, la Nintendo 64 [21], con una CPU de 64 bits a 93,75 Mhz, y la Game Boy Pocket, una versión más pequeña de la Game Boy original, que poseía un display en blanco y negro y una pantalla más grande.

En este año comienza la tercera etapa de la evolución de los motores gráficos 3D [26] y aparecen juegos como ***Duke Nukem 3D***, un videojuego de disparos en primera persona, de 3D Realms, ***Super Mario 64***, el primer plataformas en 3D, de Nintendo, ***Resident Evil***, un videojuego del género survival horror, de Capcom y ***Quake***, un videojuego de disparos en primera persona en 3D, de Id Software.

Duke Nukem 3D [24] es un videojuego con una gran variedad de niveles, formados por espacios abiertos y ambientados en calles, estaciones espaciales o incluso en ciudades sumergidas. Además cuenta con una gran cantidad de acertijos, de pasadizos y recovecos que lo hacen muy atractivo en modo multijugador.

Para el desarrollo del mismo se usó Build [43], un motor gráfico creado por Ken Silverman para 3D Realms. Al igual que Doom, Build representa los mundos a partir de formas cerradas en 2D llamadas sectores. Sin embargo, la altura y los ángulos de inclinación de estos sectores podían ser manipulados en tiempo real, sin necesidad de recalcular la información de renderizado. Esto permitía a Build representar mundos más complejos y flexibles. Por esta razón, Build es considerado un motor 2.5D.

Quake [45] fue un videojuego que revolucionó el género del 3D, pues utilizaba modelos tridimensionales para los jugadores y los monstruos y el mundo donde tenía lugar era creado con escenarios totalmente poligonales que usaban la tercera dimensión con diferentes planos de altura. A diferencia de juegos anteriores, no tenía limitaciones de ángulos, paredes o suelos. Para la creación de este videojuego se utilizó Quake Engine. Un motor gráfico creado por John Carmack y Michael Abrash.

Michael Abrash ideó algunos sistemas de optimización del rendimiento de la CPU. Uno de ellos fue reducir mediante un preprocesamiento de los escenarios, la complejidad de los mismos, mostrando exclusivamente las caras visibles de los polígonos y reduciendo de este modo los vértices totales. Otro de los sistemas utilizados, consistía en el uso y gestión del Z-buffering[36].

Esta técnica consistía en separar por medio de pasillos, salas de gran tamaño para así poder ocultar objetos y optimizar la carga del procesador.

John Carmack también introdujo avances como el sistema QuakeWorld, que mejoraba las capacidades multijugador del juego.

Figura 35: Duke Nukem 3D

Figura 36: Quake

En el año siguiente llegan al mercado grandes juegos como ***Final Fantasy VII***, para PlayStation, de Squaresoft, ***Grand Theft Auto*** de ASC Games, ***Age of Empires***, un juego de estrategia desarrollado por Ensemble Studios y publicado por Microsoft, y ***Quake II*** un videojuego desarrollado por Id Software.

- ***Quake II***: [44] Tras el éxito alcanzado por ***Quake***, Id Software decide sacar ***Quake II***, un videojuego que se aparta totalmente del ambiente mítico-medieval visto en Quake. Esta vez la acción se desarrolla en el planeta Stroggos, de raza alienígena.

  El motor gráfico usado fue id Tech 2 [27]. Las dos características más llamativas del mismo fueron el soporte directo de aceleración mediante tarjeta gráfica, específicamente OpenGL y el clásico renderizado por software.

Algunos de los juegos más importantes para PC fueron desarrollados sobre este motor, ya que tenía una gran estabilidad y potencia. Entre ellos se pueden destacar ***Half Life***, ***Counter-Strike***, ***Kingpin: Life of Crime***, ***Soldier of Fortune***, ***Sin***, ***Hexen II*** y ***Heretic II***.

1998 fue un gran año para los fps. Además de la salida de ***Half-Life*** por Valve Corporation y la consagración de ***Quake II*** [44], Epic Games crea ***Unreal***, un videojuego de disparos en primera persona.

Unreal [45]**,** fue una auténtica revolución en el mundo de los videojuegos. Los escenarios rompían con la clásica estética urbana, por primera vez se observaban escenarios naturales que se acercaban al fotorealismo.

Su motor gráfico Unreal Engine fue creado por Tim Sweeney. Este constaba de un correcto renderizado de modelos, un sistema de colisiones simple pero efectivo, una buena inteligencia artificial de los enemigos, animación facial, sincronización labial, sonido en 3D mediante A3D, iluminación volumétrica y compresión de texturas.

Half-life [45]: para la creación del mismo se utilizó Goldsrc, una variante de Quake Engine. Goldsrc es motor versátil y estable que da la posibilidad de trabajar tanto con Direct3D como con OpenGL.

En 1999 comienza la cuarta etapa de la evolución de los motores gráficos con la edición de ***Quake III*** [27] y el desarrollo de Id Tech 3, un motor gráfico desarrollado por Id Software para el mismo. Este juego es destacado por ser el primer FPS satisfactorio de juego online.

Un año más tarde [21] Sony lanza al mercado su segunda consola la PS2, dotada de una CPU a 294,9 Mhz, de un lector de DVD para juegos y películas y de puertos USB. Gracias a la fama de su antecesora la PS1, al cabo de unos meses la PS2 comenzó a venderse fácilmente, tanto, que llego a convertirse en una de las consolas más vendidas de la historia. Al principio, algunos de los juegos de la PS1 no eran compatibles con la PS2 pero este problema desapareció en versiones posteriores.

En el 2001 Microsoft entra en el mercado de las consolas con el lanzamiento de la Xbox. Fue la primera en incorporar un disco duro y poseía una CPU a 733Mhz. A pesar de que tuvo bastante éxito siempre ocupó un segundo lugar entre las más vendidas pues nunca pudo superar a la PS2.

Nintendo también lanza una nueva consola de sobremesa la Game Cube, dotada de una CPU a 485 Mhz. El diseño de la misma fue todo un éxito, al igual que su presentación.

En este año también salen al mercado grandes juegos en 3D como ***Max Payne***, desarrollado para PC por la empresa finlandesa Remedy Entertainment, ***Grand Theft Auto III***, desarrollado por Rockstar North y publicado por la compañía Rockstar Games, y ***Halo: Combat Evolved***, desarrollado por Bungie Studios y publicado por Microsoft Game Studios.

Max Payne [62] es un videojuego de disparos en primera persona, que introduce el concepto de bullet time o tiempo bala. Cuando el jugador activa el tiempo bala, la acción del juego se pone en cámara lenta de forma temporal. Lo que le permite, reaccionar en tiempo real para realizar ataques más certeros o evitar las balas de los enemigos y hacer movimientos especiales. Además a medida que se avanza en el juego, el jugador va observando presentaciones que desarrollan la historia del mismo. En estas presentaciones aparecen viñetas dotadas de sonidos y de las voces de los personajes que aparecen en ellas.

Aunque ***Max Payne*** no se lanzó hasta 2001, Remedy Entertainment llevaba desarrollando el motor desde 1997. Remedy Max FX fue desarrollado desde cero como motor 3D de aceleración hardware optimizado para DirectX[9] 7.0.

Grand Theft Auto III, continuación de la saga Grand Theft Auto y primero en presentar un ambiente de juego 3-D. Este videojuego es considerado como uno de los mejores de toda la saga y uno de los más revolucionarios de la historia.

El motor gráfico que usa es el RenderWare de Criterion Software. La característica más relevante del mismo es su capacidad para reproducirse en cualquier tipo de plataforma (Windows, Mac, GameCube, Xbox, PS2, PS3, Xbox 360) gracias a la evolución que ha experimentado con el paso del tiempo.

Figura 37: Max Payne

Figura 38: Grand Theft Auto III

En 2002 juegos como ***Battlefield 1942***, ***Unreal Tournament 2003*** y ***Tron 2.0*** son lanzados al mercado.

Battlefield 1942 [63] es un videojuego de simulación bélica desarrollado por Digital Illusions CE y distribuido por Electronic Arts. El juego constaba de grandes escenarios de batalla, un gran número de armas y de vehículos que permitía al jugador planear estrategias de guerra. Este juego utilizó el motor gráfico Refractor 2.0, que fue desarrollado por Refraction Games.

Unreal Tournament 2003 [46] es un videojuego de la saga Unreal, desarrollado por Epic Games y Digital Extremes y distribuido por Atari. Este juego utiliza el motor gráfico Unreal Engine 2, una mejora de su predecesor Unreal. Este motor cuenta con física ragdoll[12], soporte para PlayStation2, GameCube y Xbox y una carga dinámica de código y contenido.

Figura 39: Battlefield 1942

Figura 40: Unreal Tournament 2003

En el 2003 nacen videojuegos como ***Max Payne2***, con un motor MAX.FX 2.0 para Windows y un motor RenderWare para PS2 y Xbox, ***Rainbow Six 3***, con un motor Unreal Engine 2, y ***Call of Duty***, con una mejora del motor de Id Softwares, Id Tech 3.

El 2004 marca el inicio de la quinta etapa de la evolución de los motores gráficos 3D, que no terminará hasta dos años más tarde. En la misma, surgen motores como Id Tech 4, usado para juegos como Doom 3 o Quake 4, CryEngine, usado en el famoso juego ***Far Cry***, y Source Engine, utilizado en ***Half-life 2***.

Id Tech 4 [27] es un motor diseñado por John Carmack, para Id Software en 1999, y usado por primera vez en el 2004 para el juego Doom 3. Este motor permitía el uso de distintas texturas: bump zapping que permitía generar falso relieve 3D en un modelo, normal zapping, para generar desplazamiento perpendicular sobre un polígono y specular highlight, para definir la cantidad de brillo de una superficie.

CryEngine [27] es un motor de juego creado por la empresa alemana Crytek. Este motor fue utilizado para el juego Far Cry que hizo del él, un completo éxito gracias a los gráficos ultrarrealistas, las iluminaciones calculadas en tiempo real, la libertad de los movimientos del personaje y la inteligencia artificial fácilmente programable.

Source fue desarrollado por Valve Corporation para las plataformas Windows (32 y 64 bits), Mac OS X, Xbox, Xbox 360, y PlayStation 3. Este motor fue utilizado en videojuegos como Counter-Strike: Source y Half-Life 2.

Durante estos años Nintendo lanza la Nintendo DS, una consola compuesta por dos pantallas, una de ellas táctil, un micrófono con reconocimiento de voz y una conexión wifi para jugar online. Más tarde aparecería un rediseño de la misma, la Nintendo DS Lite.

Ante el éxito de las consolas portátiles de Nintendo, Sony decide lanzar la suya, la PSP. Una consola que contaba con una pantalla enorme con una gran calidad y un formato llamado UMD que permitía ver películas en ella.

En 2005 Microsoft presenta su segunda consola la Xbox 360, con un procesador Xenon y 3 núcleos de procesamiento paralelos de 3,2 GHz cada uno.

En 2006 llega al mercado la tercera consola de sobremesa de Sony la PlayStation 3 y una consola revolucionaria de Nintendo, la Wii.

Este año también es destacado por el comienzo de la sexta etapa de los motores gráficos, y con ella un gran salto tecnológico. Los motores gráficos más destacados desde esta fecha a día de hoy son: **Cry Engine 2**, **Unreal Engine 3**, **Cry Engine 3**, **IW Engine** y **RAGE**.

Cry Engine 2 [27] es el motor más avanzado de su generación. Dio un salto tecnológico tan grande que muchos usuarios de PC no tuvieron más remedio que cambiar la tarjeta gráfica. Entre las mejoras de este motor destaca un entorno 3D totalmente realista, con objetos variados y destructibles y una iluminación global y volumétrica. Algunos de los juegos desarrollados con el mismo son Crysis y Soldier of Fortune Payback.

Unreal Engine 3 [27], es la tercera generación de Unreal, diseñado para PC y con soporte para DirectX 9, DirectX 10, Xbox 360 y PlayStation 3. Este motor es capaz de desarrollar grandes juegos con escenarios 3D. Para ello, utiliza técnicas como la iluminación HDR de 64 bits, renderización multinúcleo, efectos de postprocesado en tiempo real y efectos volumétricos entre otros. También permite la optimización del desarrollo de los juegos.

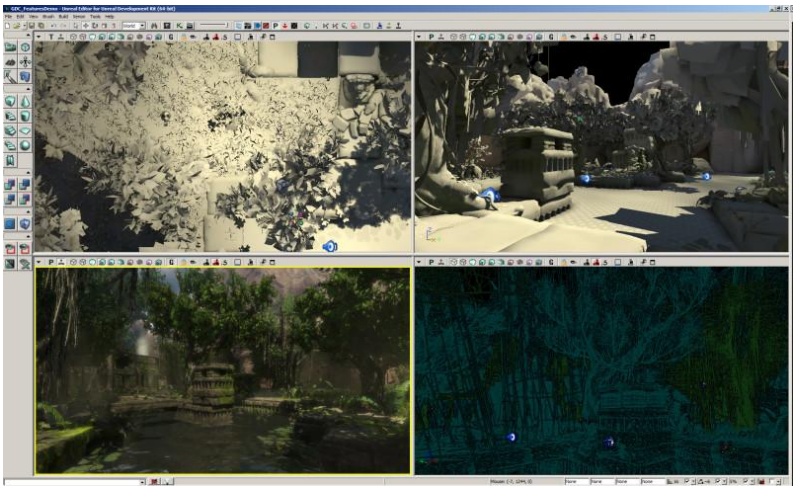


Figura 41: Editor WYSIWYG

El editor es WYSIWYG. Este editor permite crear un modelo 3D con un tamaño, una textura y una iluminación determinada de la manera más sencilla, sólo es necesario pinchar y arrastrar. Además posee el mejor visor de bibliotecas que integra modelos, texturas, materiales, efectos especiales y animaciones.

CryEngine 3 [27] es un motor de videojuego premiado por ofrecer la mejor simulación en tiempo real. CryEngine 3 es usado en juegos como: ASTA, ArcheAge, Cabal 2, Crysis 2 y Warface.

IW Engine es el motor que usan juegos como ***Modern Warefare*** y ***Black Ops***. Los gráficos y las animaciones que proporciona son geniales, al igual que la inteligencia artificial de los enemigos de los mismos. Pero sin duda lo más destacable es el juego online que proporciona.

RAGE es un motor polifuncional en muchos sentidos, ya que puede manejar grandes mundos abiertos con muchos interiores, inteligencia artificial compleja, efectos de clima en tiempo real y distintos tipos de interfaces. Uno de los juegos que uso Rage fue ***GTA IV***.

2.4.- Historia de los teléfonos móviles

En 1947 [33], en los Bell Laboratories, nace la idea de comunicaciones móviles utilizando una red celular. Por aquel entonces, dicha idea no se podía llevar a cabo, a causa de la dificultad de concesión de espectro de radio por parte de las autoridades. Pero en 1960 comienza a ser factible y dos empresas, Motorola y Bell Labs, deciden competir por el desarrollo de la misma.

El 3 de Abril de 1973 Martin Cooper, director corporativo de Investigación y Desarrollo de Motorola, finaliza el primer prototipo de teléfono móvil de la historia, el DynaTAC y decide comunicárselo a su competidor Joel Engel, el responsable del departamento de pesquisa de Bell Labs, a través del mismo. Desde entonces Martin Cooper pasaría a la historia por ser el primero en realizar una llamada a través de un teléfono móvil.

En el año 1983 DynaTAC obtiene la licencia comercial y el primer teléfono móvil analógico conocido con el nombre de Motorola Dyna TAC 8000X, es lanzado al mercado. La baja calidad de las conversaciones telefónicas, la poca duración de la batería y su gran peso y tamaño, hacían de él, un aparato de limitadas prestaciones técnicas. Sin embargo, era un lujo orientado a muy pocos sectores y de un elevado coste.

El Motorola Dyna TAC 8000X fue un éxito de ventas inesperado, solo en el primer año se vendieron un total de 900.000 unidades.

La primera generación

A partir de este momento comienza la primera generación de los teléfonos móviles, que durará hasta el año 1989 con la aparición de MicroTAC, un teléfono móvil analógico con un diseño innovador.

La segunda generación

La segunda generación comienza en 1990 en Estados Unidos con la primera llamada desde un teléfono móvil digital.

A partir de ese momento, los teléfonos analógicos van desapareciendo poco a poco, abriendo paso a la digitalización de las comunicaciones y los beneficios de la misma.

Las comunicaciones digitales proporcionaban al usuario una mejor calidad en las conversaciones y un aumento del nivel de seguridad. Además el precio del terminal fue reducido por la simplificación en la fabricación de mismo.

En los siguientes años se producirán grandes avances tecnológicos en el campo de la telefonía móvil que ampliará el número de utilidades de estos dispositivos; captura de fotos y videos, radio y juegos.

En 1997 aparece uno de los primeros juegos para teléfonos móviles, el **Snake,** de Nokia. A pesar de la sencillez del mismo, el juego se convierte en un gran éxito.

La tercera generación

En esta generación aparecen los primeros smartphones y con ellos los distintos sistemas operativos móviles, entre los que destaca Android.

2.5.- Android

Android, es un sistema operativo basado en GNU/Linux [17] y desarrollado por Android Inc., que finalmente fue comprado por Google en 2005. En 2007, Android es lanzado al mercado bajo la licencia Apache, una licencia de software libre que permite al desarrollador el acceso completo al software del sistema, la modificación e incluso la distribución del mismo. De este modo, los desarrolladores pueden escribir aplicaciones para extender la funcionalidad de los dispositivos y ponerlas de forma gratuita o de pago en el mercado de Android.

El lenguaje en el que están escritas estas aplicaciones es Java[22], orientado a objetos, disponiendo además de una máquina virtual llamada Dalvik[8] con compilación en tiempo de ejecución y de un conjunto de librerías escritas en C[5]/C++[6], entre las que se destacan una API gráfica OpenGL ES 1.0 3D, un motor de renderizado WebKit[35] y un motor gráfico SGL.

2.5.1.- Motores gráficos para Android

Los motores gráficos 2D más destacados de Android son los siguientes:

- **Rokon** es un motor gráfico fácil de usar, bien documentado, con un gran número de ejemplos, tutoriales básicos y bastante flexible. Un ejemplo de aplicación para este motor es el Drop Block, un juego con un objetivo bastante claro, eliminar todos los bloques de la pantalla dejando como último bloque el de la superficie a cuadros.

- **Libgdx**: Es un framework[16] multiplataforma para el desarrollo de juegos en Windows, Linux y Android, escrito en Java y C/C++. El lenguaje en C/C++ se empleará para dar soporte y rendimiento a tareas relacionadas con el uso de la física y procesamiento de audio. El motor gráfico Libgdx permite a los desarrolladores de aplicaciones escribir, probar y depurar las mismas en el PC.

  "*Las herramientas que nos brinda Libgdx son las siguientes:*
  Un framework que nos permitirá manejar el ciclo de vida de la aplicación.
  Un módulo de gráficos que nos proporciona una forma de dibujar objetos en la pantalla.
  Un módulo de audio para reproducir música y efectos de audio.
  Un módulo de entrada para recibir toda la información del usuario proveniente del mouse, teclado, pantalla táctil, acelerómetro, etc.
  Un módulo de I/O para leer y escribir datos como texturas, mapas o archivos de configuración.
  " [31]

- **AndEngine** [32]: es un motor gráfico 2D para Android de código abierto. Es una implementación 2D de OpenGL para Android, por lo tanto, el lenguaje de programación utilizado para la implementación de los videojuegos será Java. El principal inconveniente de este motor es la falta de documentación oficial disponible.

  Algunos de los conceptos básicos del motor son; el BaseGameActivity o raíz del juego, que contiene el motor y crea la vista donde se dibujará posteriormente, el Engine o motor interno, que se encargará de dibujar y actualizar los objetos de la escena, un objeto cámara, la clase Scene que es un contenedor para todos los objetos que se van a dibujar en la escena, una entidad…

- **Angle** [47] es un motor gráfico que destaca por el rendimiento y la velocidad, permitiendo crear aplicaciones de más de 60 fps. El motor es bastante sencillo de usar y posee un conjunto de tutoriales que enseñan a programar en el mismo paso a paso. El lenguaje de programación que utiliza es Java.

Los motores gráficos 3D más destacados de Android son los siguientes:

- **jPCT-AE** [48] : Es un motor 3D de código abierto para el diseño de videojuegos en Android. El lenguaje de programación sobre el que trabaja es Java y ofrece al desarrollador la posibilidad de hacer videojuegos de una forma sencilla y rápida. Uno de los juegos implementados a partir de este motor es Alien Runner.

Figura 42: Alien Runner

- **Forget3D**: Es un framework multiplataforma para el desarrollo de juegos en las plataformas Android, Win32 y WinCE.

- **Mages**: Este motor facilita el desarrollo de juegos multijugador para Internet [49].

2.6.- OpenGL

En los años 1980 [51] se pretendía desarrollar un software que fuera compatible con un amplio rango de hardware gráfico, dicha tarea era bastante compleja, pues había que tratar con interfaces muy diversos y con drivers específicos de cada hardware.

Finalmente y tras mucho trabajo, en 1992, la empresa Silicon Graphics Inc. lanza el estándar OpenGL, una especificación estándar que define una API para escribir aplicaciones que produzcan gráficos 2D y 3D, desarrollada por Mark Segal y Kurt Akeley. OpenGL comienza a usarse en el desarrollo de videojuegos.

Junto con el lanzamiento de OpenGL, se funda el Open Architecture Review Board (ARB), un conjunto de empresas interesadas en dicha especificación cuyo objetivo era revisar el estándar y ampliarlo. OpenGL estuvo bajo el control de ARB hasta el año 2006 donde el Grupo Khronos se hace con el control del mismo. Gracias a estas empresas OpenGL se convierte en un API con un amplio rango de posibilidades.

OpenGL es un documento que describe un conjunto de funciones que permiten desarrollar escenas tridimensionales complejas a partir de la unión de primitivas más simples. OpenGL ofrece distintos tipos de primitivas o unidades básicas de dibujado.

Una vez establecidas las primitivas utilizadas OpenGL procederá a convertirlas en píxeles por medio de una pipeline gráfica[29] conocida como Máquina de estados de OpenGL.

OpenGL es una API basada en procedimientos de bajo nivel, es decir, el programador deberá dar instrucciones precisas para el renderizado de la escena y conocer en profundidad el modo de funcionamiento de la pipeline gráfica.

A partir de la documentación de OpenGL los fabricantes de hardware crean variantes simplificadas como OpenGL ES, una especificación diseñada para teléfonos móviles, PDAs y consolas por el Grupo Khronos.

2.6.1.- OpenGL ES

OpenGL ES es una API para gráficos 3D utilizada oficialmente en el sistema operativo Symbian OS, en la plataforma para dispositivos móviles Android y en el sistema operativo iOS para el iPhone.

Existen varias versiones de la especificación OpenGL ES.

- Versión 1.0: Con la creación de esta versión se introdujeron algunos cambios en el código de OpenGL. El más significativo fue la creación de interfaces de coma fija para todos sus métodos.

  Este cambio fue bastante útil debido a la carencia de unidad de coma flotante o FPU [15] de algunos procesadores integrados o a la limitada capacidad de cómputo de los mismos. También destacan otros cambios como la eliminación de la semántica de las llamadas glBegin-glEnd para representar primitivas.

- Versión 1.1: esta versión es compatible con la anterior e introduce algunos cambios sobre la misma como el soporte obligatorio de multi-texturas y generación automática de mipmaps.

- Versión 2.0: está versión no es compatible con OpenGL ES 1.1.

CAPÍTULO 3.- DESARROLLO

3.1.- Casos de uso

Un diagrama de casos de uso es similar a un diagrama de comportamiento. En él, se describen las actividades que una entidad externa o actor, demanda al sistema. Al tratarse de un motor gráfico, todas las funcionalidades del mismo estarán relacionadas con el renderizado de una escena en 3D y el actor que demandará dichas actividades al sistema será un programador.

Una vez realizado el diagrama se procederá a reunir la información en unas tablas con los siguientes campos:

- **Identificador:** Identificación del caso de uso. Los casos de uso se identificarán mediante la nomenclatura CU, donde XX corresponderá con un número del 01 al 05.
- **Nombre:** Título que recibe un caso de uso.
- **Actores**: usuarios que intervienen en la realización del caso de uso.
- **Precondiciones**: Condiciones bajo las cuales se da el caso de uso.
- **Postcondiciones**: Indican el estado del sistema después de la ejecución del caso de uso.
- **Descripción**: Breve explicación de los detalles del caso de uso.

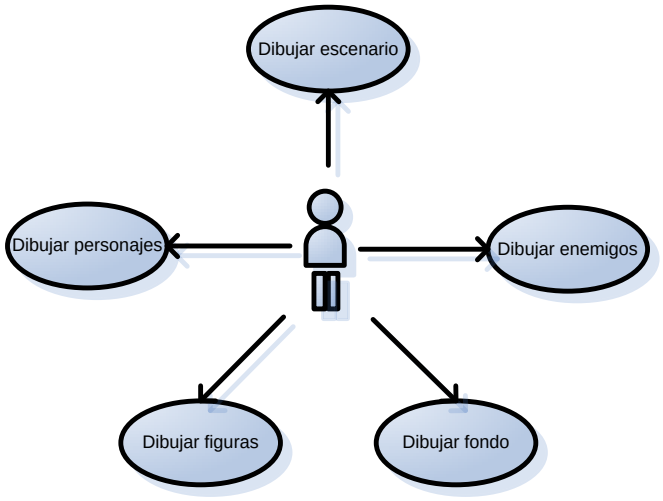


Figura 43: Diagrama de casos de uso

IDENTIFICADOR CU_01	
Nombre	Dibujar figuras
Actores	Programador
Precondiciones	- Rellenar un vector con las figuras que componen el escenario
Poscondiciones	- Dibujar las figuras en la posición indicada
Descripción	El programador rellenará un vector con las figuras deseadas y el sistema se encargará de dibujar dichas figuras en la posición indicada.

IDENTIFICADOR CU_02	
Nombre	Dibujar escenario
Actores	Programador
Precondiciones	- Construir una matriz de tres dimensiones compuesta por enteros, que represente el escenario a construir. - La distancia de las figuras con respecto al punto de enfoque de la cámara es menor o igual a una distancia d establecida.
Poscondiciones	- Dibujar el escenario de juego
Descripción	El sistema se encarga de dibujar el escenario 3D sobre el cual se desarrollará la lógica del juego.

IDENTIFICADOR CU_03	
Nombre	Dibujar Personajes
Actores	Programador
Precondiciones	- Indicar la posición del personaje
Poscondiciones	- Dibujar el personaje en la posición actual con su movimiento correspondiente.
Descripción	El sistema se encarga de dibujar el personaje principal del juego y actualizarlo en función del movimiento del mismo.

IDENTIFICADOR CU_04	
Nombre	Dibujar Enemigos
Actores	Programador
Precondiciones	- Indicar la posición de los enemigos. - La distancia de los enemigos con respecto al punto de enfoque de la cámara es menor o igual a una distancia d establecida.
Poscondiciones	- Dibujar los enemigos en la posición actual con su movimiento correspondiente.
Descripción	El sistema se encarga de dibujar el personaje principal del juego y actualizarlo en función del movimiento del mismo.

IDENTIFICADOR CU_05	
Nombre	Dibujar Fondo
Actores	Programador
Precondiciones	- Indicar la posición del personaje principal y de la cámara
Poscondiciones	- Dibujar el fondo en la posición indicada
Descripción	El sistema se encarga de dibujar el fondo del escenario.

3.2.- Diagrama de secuencia

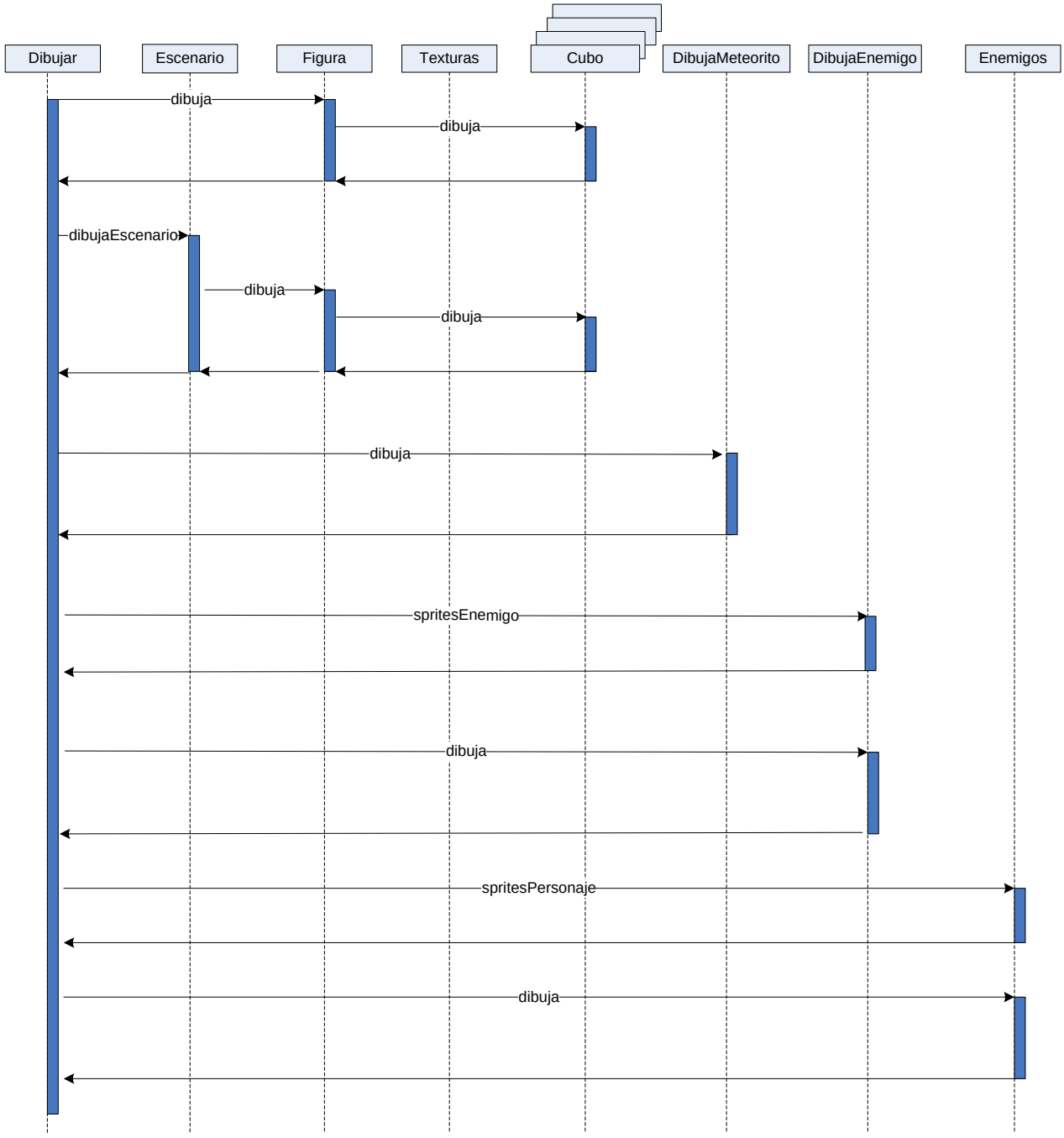


Figura 44: Diagrama de secuencia dibujar

3.3.- Análisis de requisitos

3.3.1.- Requisitos de usuario

En este apartado se especifican los principales requisitos de usuario del proyecto, los cuales proporcionarán una visión detallada del comportamiento externo del sistema, evitando en la medida de lo posible las características de diseño del mismo.

Dentro de los requisitos de usuario se hará una diferenciación entre los requisitos de capacidad, que representan una necesidad o servicio requerido por el usuario, y los requisitos de restricción, que representan una restricción o imposición para el usuario.

Para su especificación se hará uso de unas tablas con los siguientes campos:

- **Identificador:** Identificación del requisito. Los requisitos de usuario de capacidad se identificarán mediante la nomenclatura RUC_XX, donde XX corresponderá con un número del 01 al 17 que indicará el número del requisito. Los requisitos de restricción se identificarán mediante la nomenclatura RUR_XX, donde XX corresponderá con un número del 18 al 20.

- **Descripción:** Breve explicación de los detalles del requisito.

- **Fuente:** A partir de que caso de uso surge el requisito de usuario.

- **Claridad:** Se indicará la claridad del requisito por medio de una escala del 1 al 5.

- **Prioridad:** Urgencia con la que debe ser resuelto el requisito, alta media o baja.

- **Necesidad:** Indica si el requisito es esencial u opcional.

- **Verificable:** Indica si se puede verificar el requisito con una prueba.

- **Estabilidad:** El requisito cambia a lo largo del desarrollo del proyecto.

- **Criterio de cumplimiento:** Indica si se ha llevado a cabo el requisito.

IDENTIFICADOR RUC_01	
Descripción	Dibujar una esfera
Fuente	CU_01
Claridad	3
Prioridad	Media
Necesidad	Opcional
Verificable	Si
Estabilidad	Variable
Criterio de cumplimiento	El usuario podrá dibujar una esfera indicando el radio y el material que se va a utilizar para su diseño.

IDENTIFICADOR RUC_02	
Descripción	Dibujar un triángulo y un cuadrado
Fuente	CU_01
Claridad	4
Prioridad	Media
Necesidad	Opcional
Verificable	Si
Estabilidad	Variable
Criterio de cumplimiento	Para el diseño de un triángulo o un cuadrado, el usuario deberá definir las dimensiones (ancho y alto) y el material del mismo.

IDENTIFICADOR RUC_03	
Descripción	Dibujar un cubo y una pirámide
Fuente	CU_01
Claridad	3
Prioridad	Media
Necesidad	Opcional
Verificable	Si
Estabilidad	Variable
Criterio de cumplimiento	Para el diseño de un cubo o una pirámide el usuario deberá definir las dimensiones (ancho, alto y largo) y el material del mismo.

IDENTIFICADOR RUC_04	
Descripción	Dibujar un triángulo y un cuadrado a color
Fuente	CU_01
Claridad	4
Prioridad	Media
Necesidad	Opcional
Verificable	Si
Estabilidad	Variable
Criterio de cumplimiento	Para el diseño de un triángulo o un cuadrado a color, el usuario deberá definir las dimensiones del mismo (ancho y alto).

IDENTIFICADOR RUC_05	
Descripción	Dibujar un cubo y una pirámide a color
Fuente	CU_01
Claridad	4
Prioridad	Media
Necesidad	Opcional
Verificable	Si
Estabilidad	Variable
Criterio de cumplimiento	Para el diseño de un cubo o una pirámide a color, el usuario deberá definir las dimensiones del mismo (ancho, alto y largo).

IDENTIFICADOR RUC_06	
Descripción	Dibujar un triángulo y un cuadrado con textura
Fuente	CU_01
Claridad	4
Prioridad	Alta
Necesidad	Esencial
Verificable	Si
Estabilidad	Variable
Criterio de cumplimiento	Para el diseño de un triángulo o un cuadrado con textura, el usuario deberá definir las dimensiones (ancho y alto) y la textura que se mapeará sobre la superficie del mismo.

IDENTIFICADOR RUC_07	
Descripción	Dibujar un cubo y una pirámide con textura
Fuente	CU_01
Claridad	4
Prioridad	Alta
Necesidad	Esencial
Verificable	Si
Estabilidad	Variable
Criterio de cumplimiento	Para el diseño de un cubo o una pirámide con textura, el usuario deberá definir las dimensiones (ancho, alto y largo) y la textura que se mapeará sobre la superficie del mismo.

IDENTIFICADOR RUC_08	
Descripción	Dibujar un triángulo y un cuadrado con una textura y el fondo transparente
Fuente	CU_01
Claridad	4
Prioridad	Alta
Necesidad	Esencial
Verificable	Si
Estabilidad	Variable
Criterio de cumplimiento	Para el diseño de un triángulo o un cuadrado con textura y con el fondo transparente, el usuario deberá definir las dimensiones (ancho y alto) y la textura fuente y máscara, que se mapearán sobre la superficie del mismo.

IDENTIFICADOR RUC_09	
Descripción	Dibujar un cubo y una pirámide con una textura y el fondo transparente
Fuente	CU_01
Claridad	4
Prioridad	Alta
Necesidad	Esencial
Verificable	Si
Estabilidad	Variable
Criterio de cumplimiento	Para el diseño de un cubo o una pirámide con textura y con el fondo transparente, el usuario deberá definir las dimensiones (ancho, alto y largo) y la textura fuente y máscara que se mapeará sobre la superficie del mismo.

IDENTIFICADOR RUC_10	
Descripción	Dibujar un cubo con múltiples texturas
Fuente	CU_01
Claridad	4
Prioridad	Alta
Necesidad	Opcional
Verificable	Si
Estabilidad	Variable
Criterio de cumplimiento	Para el diseño de un cubo con una textura distinta para cada una de sus caras, el usuario deberá definir las dimensiones (ancho, alto y largo) y las texturas.

IDENTIFICADOR RUC_11	
Descripción	Dibujar el escenario
Fuente	CU_02
Claridad	3
Prioridad	Alta
Necesidad	Esencial
Verificable	Si
Estabilidad	Estable
Criterio de cumplimiento	El usuario indicará a través de una matriz tridimensional, la posición de las figuras que desea dibujar. El sistema dibujará cada figura en la posición indicada.

IDENTIFICADOR RUC_12	
Descripción	Iluminación de la escena
Fuente	CU_02
Claridad	4
Prioridad	Media
Necesidad	Opcional
Verificable	Si
Estabilidad	Variable
Criterio de cumplimiento	El usuario indicará las características de iluminación de la escena y el sistema se encargará de iluminar la misma.

IDENTIFICADOR RUC_13	
Descripción	Dibujar los sprites del personaje principal
Fuente	CU_03
Claridad	4
Prioridad	Alta
Necesidad	Esencial
Verificable	Si
Estabilidad	Variable
Criterio de cumplimiento	El sistema se encargará de dibujar el personaje principal.

IDENTIFICADOR RUC_14	
Descripción	Actualizar los sprites del personaje principal
Fuente	CU_03
Claridad	4
Prioridad	Alta
Necesidad	Esencial
Verificable	Si
Estabilidad	Variable
Criterio de cumplimiento	El sistema se encargará de actualizar los sprites del personaje principal para simular el movimiento de los mismos.

IDENTIFICADOR RUC_15	
Descripción	Dibujar los sprites de los enemigos
Fuente	CU_04
Claridad	4
Prioridad	Alta
Necesidad	Esencial
Verificable	Si
Estabilidad	Variable
Criterio de cumplimiento	El sistema se encargará de dibujar los enemigos.

IDENTIFICADOR RUC_16	
Descripción	Actualizar los sprites de los enemigos
Fuente	CU_04
Claridad	4
Prioridad	Alta
Necesidad	Esencial
Verificable	Si
Estabilidad	Variable
Criterio de cumplimiento	El sistema se encargará de actualizar los sprites de los enemigos para simular el movimiento de los mismos.

IDENTIFICADOR RUC_17	
Descripción	Dibujar el fondo
Fuente	CU_05
Claridad	4
Prioridad	Alta
Necesidad	Esencial
Verificable	Si
Estabilidad	Variable
Criterio de cumplimiento	Para dibujar el fondo el usuario deberá indicar las dimensiones y la textura del mismo. El sistema dibujará el fondo simulando un movimiento del personaje principal.

IDENTIFICADOR RUR_18	
Descripción	Formatos y dimensiones de texturas
Fuente	CU_01, CU_02, CU_03, CU_04, CU_05
Claridad	5
Prioridad	Alta
Necesidad	Esencial
Verificable	Si
Estabilidad	Variable
Criterio de cumplimiento	Las dimensiones de las texturas deberán ser potencia de dos, su ancho deberá de ser igual a su alto y su formato .png.

IDENTIFICADOR RUR_19	
Descripción	Color de las figuras
Fuente	CU_01
Claridad	5
Prioridad	Alta
Necesidad	Esencial
Verificable	Si
Estabilidad	Variable
Criterio de cumplimiento	En caso de que el usuario quiera dibujar figuras a color, no deberá iluminar la escena.

IDENTIFICADOR RUR_20	
Descripción	Figuras
Fuente	CU_01
Claridad	5
Prioridad	Alta
Necesidad	Esencial
Verificable	Si
Estabilidad	Variable
Criterio de cumplimiento	El usuario deberá escoger si desea dibujar el videojuego con figuras iluminadas, figuras a color o figuras con texturas.

3.3.2.- Requisitos software

En este apartado se especifican los principales requisitos software del proyecto, los cuales proporcionarán una visión detallada de los requerimientos del sistema.

Dentro de los requisitos software se hará una diferenciación entre los requisitos funcionales, que especifican acciones que el sistema debe ser capaz de realizar de una manera clara y libre de ambigüedades, y los requisitos no funcionales de rendimiento, interfaz, operación, recursos, comprobación, documentación, seguridad, calidad y mantenimiento.

Para la especificación de estos requisitos se utilizarán los siguientes campos:

- **Identificador:** Identificación del requisito.

  - Los requisitos funcionales software se identificarán mediante la nomenclatura RSF_XX, donde XX corresponderá con un número del 01 al 17 que indicará el número del requisito.
  - El requisito de interfaz se identificará mediante la nomenclatura RSI_01
  - Los requisitos software de recursos se identificarán mediante la nomenclatura RSR_XX, donde XX corresponderá con los números 01 y 02.
  - El requisito de rendimiento se identificará mediante la nomenclatura RSRD_ XX, donde XX corresponderá con los números 01 y 02.

- **Descripción:** Breve explicación de los detalles del requisito.
- **Fuente:** A partir de que requisito de usuario surge el requisito software.
- **Claridad:** Se indicará la claridad del requisito por medio de una escala del 1 al 5.
- **Prioridad:** Urgencia con la que debe ser resuelto el requisito, alta media o baja.
- **Necesidad:** Indica si el requisito es esencial u opcional.
- **Verificable:** Indica si se puede verificar el requisito con una prueba.
- **Estabilidad:** El requisito cambia a lo largo del desarrollo del proyecto.

IDENTIFICADOR RSF_01	
Descripción	Diseñar las figuras mediante la herramienta OpenGL ES 1.0
Fuente	RUC_01, RUC_02, RUC_03, RUC_04, RUC_05, RUC_06, RUC_07, RUC_08, RUC_09, RUC_10
Claridad	5
Prioridad	Alta
Necesidad	Esencial
Verificable	Si
Estabilidad	Estable

IDENTIFICADOR RSF_02	
Descripción	Establecer los vértices de las figuras
Fuente	RUC_01, RUC_02, RUC_03, RUC_04, RUC_05, RUC_06, RUC_07, RUC_08, RUC_09, RUC_10
Claridad	4
Prioridad	Alta
Necesidad	Esencial
Verificable	Si
Estabilidad	Estable

IDENTIFICADOR RSF_03	
Descripción	Establecer los índices para la formación de caras.
Fuente	RUC_02, RUC_03, RUC_04, RUC_05, RUC_06, RUC_07, RUC_08, RUC_09
Claridad	4
Prioridad	Alta
Necesidad	Esencial
Verificable	Si
Estabilidad	Estable

IDENTIFICADOR RSF_04	
Descripción	Habilitar el uso de luces y establecer el tipo de sombreado de las superficies.
Fuente	RUC_01, RUC_02, RUC_03, RUC_12
Claridad	5
Prioridad	Media
Necesidad	Opcional
Verificable	Si
Estabilidad	Estable

IDENTIFICADOR RSF_05	
Descripción	Configurar las luces y las propiedades de los materiales.
Fuente	RUC_01, RUC_02, RUC_03, RUC_12
Claridad	5
Prioridad	Media
Necesidad	Opcional
Verificable	Si
Estabilidad	Estable

IDENTIFICADOR RSF_06	
Descripción	Configurar, en una matriz, los colores en los vértices de las figuras y aplicar dichos colores a las mismas.
Fuente	RUC_04, RUC_05
Claridad	5
Prioridad	Media
Necesidad	Esencial
Verificable	Si
Estabilidad	Estable

IDENTIFICADOR RSF_07	
Descripción	Generar un array[2] de datos que represente la textura indicada por el usuario con las componentes RGB del color de la misma. Mapear la textura en la superficie del objeto deseado.
Fuente	RUC_06, RUC_07, RUC_08, RUC_09, RUC_10
Claridad	4
Prioridad	Alta
Necesidad	Esencial
Verificable	Si
Estabilidad	Variable

IDENTIFICADOR RSF_08	
Descripción	Recorrer una matriz de enteros tridimensional y en función del contenido de la misma dibujar una figura u otra en la posición indicada, por medio de transformaciones de la cámara.
Fuente	RUC_11
Claridad	3
Prioridad	Alta
Necesidad	Esencial
Verificable	Si
Estabilidad	Variable

IDENTIFICADOR RSF_09	
Descripción	Si el usuario desea crear un escenario compuesto de figuras con texturas, deberá introducir en un vector, las texturas que desea mapear sobre las superficies de las figuras.
Fuente	RUC_11
Claridad	3
Prioridad	Alta
Necesidad	Opcional
Verificable	Si
Estabilidad	Variable

IDENTIFICADOR RSF_10	
Descripción	Aplicar las técnicas de OpenGL ES de transparencia o blending y de enmascaramiento o masking, para dibujar las figuras con una textura dada y un fondo transparente.
Fuente	RUC_08, RUC_09
Claridad	4
Prioridad	Media
Necesidad	Opcional
Verificable	Si
Estabilidad	Variable

IDENTIFICADOR RSF_11	
Descripción	Obtener la imagen formada por la unión de 12 subimágenes, que representan el movimiento del personaje en cuestión, y recortarlas en 12 partes iguales, de tal manera, que en cada una de ellas aparezca un movimiento individual.
Fuente	RUC_13, RUC_15
Claridad	5
Prioridad	Alta
Necesidad	Esencial
Verificable	Si
Estabilidad	Estable

IDENTIFICADOR RSF_12	
Descripción	Obtener la imagen formada por la unión de 12 máscaras correspondientes al movimiento del personaje en cuestión, y recortarlas en 12 partes iguales, de tal manera, que en cada una de ellas aparezca un movimiento individual.
Fuente	RUC_13, RUC_15
Claridad	5
Prioridad	Alta
Necesidad	Esencial
Verificable	Si
Estabilidad	Estable

IDENTIFICADOR RSF_13	
Descripción	Crear un array de 4 filas y 3 columnas, y en cada posición del mismo crear un cuadrado. Aplicar las propiedades de transparencia, dibujar sobre ellos las distintas máscaras obtenidas en RSF_12, aplicar las propiedades de enmascarado y dibujar sobre ellos las distintas texturas obtenidas en RSF_11.
Fuente	RUC_13, RUC_15
Claridad	5
Prioridad	Alta
Necesidad	Esencial
Verificable	Si
Estabilidad	Estable

IDENTIFICADOR RSF_14	
Descripción	Actualizar la posición del movimiento por medio de dos variables, filas y columnas. De tal manera que cuando el personaje cambie de movimiento, se actualicen las filas y cuando continúe con el mismo, se actualicen las columnas.
Fuente	RUC_14, RUC_16
Claridad	5
Prioridad	Alta
Necesidad	Esencial
Verificable	Si
Estabilidad	Variable

IDENTIFICADOR RSF_15	
Descripción	Dibujar el cuadrado que corresponda con la posición del array con las filas y las columnas mencionadas en RSF_14.
Fuente	RUC_14, RUC_16
Claridad	5
Prioridad	Alta
Necesidad	Esencial
Verificable	Si
Estabilidad	Variable

IDENTIFICADOR RSF_16	
Descripción	Crear una figura con unas dimensiones dadas que represente el fondo.
Fuente	RUC_17
Claridad	5
Prioridad	Alta
Necesidad	Esencial
Verificable	Si
Estabilidad	Variable

IDENTIFICADOR RSF_17	
Descripción	Dibujar el fondo de tal manera que simule velocidad.
Fuente	RUC_17
Claridad	5
Prioridad	Alta
Necesidad	Esencial
Verificable	Si
Estabilidad	Variable

IDENTIFICADOR RSI_01	
Descripción	Cargar imágenes tanto para las texturas de los objetos como para los sprites cuyas dimensiones sean potencia de dos y su ancho sea igual a su alto
Fuente	RUR_12
Claridad	5
Prioridad	Alta
Necesidad	Esencial
Verificable	Si
Estabilidad	Estable

IDENTIFICADOR RSR_01	
Descripción	Desarrollo del juego mediante las herramientas: Eclipse, OpenGL ES 1.0, Android y un teléfono móvil con un sistema operativo Android con una versión 2.2 o inferior y con una RAM bastante alta para analizar el rendimiento de la aplicación
Claridad	5
Prioridad	Alta
Necesidad	Esencial
Verificable	No
Estabilidad	Estable

IDENTIFICADOR RSR_02	
Descripción	Ejecución del juego mediante un sistema operativo Android
Claridad	5
Prioridad	Alta
Necesidad	Esencial
Verificable	Si
Estabilidad	Estable

IDENTIFICADOR RSRD_01	
Descripción	Cálculo y análisis constante del número de fps. El número de fps debe de ser mayor o igual a 24
Claridad	5
Prioridad	Alta
Necesidad	Esencial
Verificable	Si
Estabilidad	Estable

IDENTIFICADOR RSRD_02	
Descripción	Mejorar el rendimiento de la aplicación dibujando los elementos del escenario y los enemigos que se encuentren a una distancia menor o igual a 8.5f.
Fuente	RUR_12
Claridad	5
Prioridad	Alta
Necesidad	Esencial
Verificable	Si
Estabilidad	Estable

CAPÍTULO 4.- IMPLEMENTACIÓN

4.1.- Fase de Diseño

En la fase de diseño se analizarán las necesidades del proyecto dividendo el mismo en clases, que se estudiarán posteriormente de manera exhaustiva, mediante un diagrama que sirva para ver la interacción y la relación de unas con otras.

4.1.1.- Diagrama de clases

El diagrama de clases es utilizado para describir las clases en las que está dividido el sistema, las relaciones existentes entre unas y otras, como pueden ser las relaciones de herencia o de visibilidad, y los atributos y métodos más importantes dentro de cada una de ellas.

Al tratarse de una librería para el diseño de un videojuego de plataformas en 3D, muchas de las clases que forman el sistema contendrán métodos de tipo abstracto, para que el usuario los implemente a su gusto más adelante. Las clases de la parte de la lógica, es decir, las que implementará el usuario del sistema al utilizar el mismo, aparecerán en color azul claro y en su interior contendrán los métodos abstractos que deben de ser implementados de manera obligatoria, para evitar que se produzcan errores.

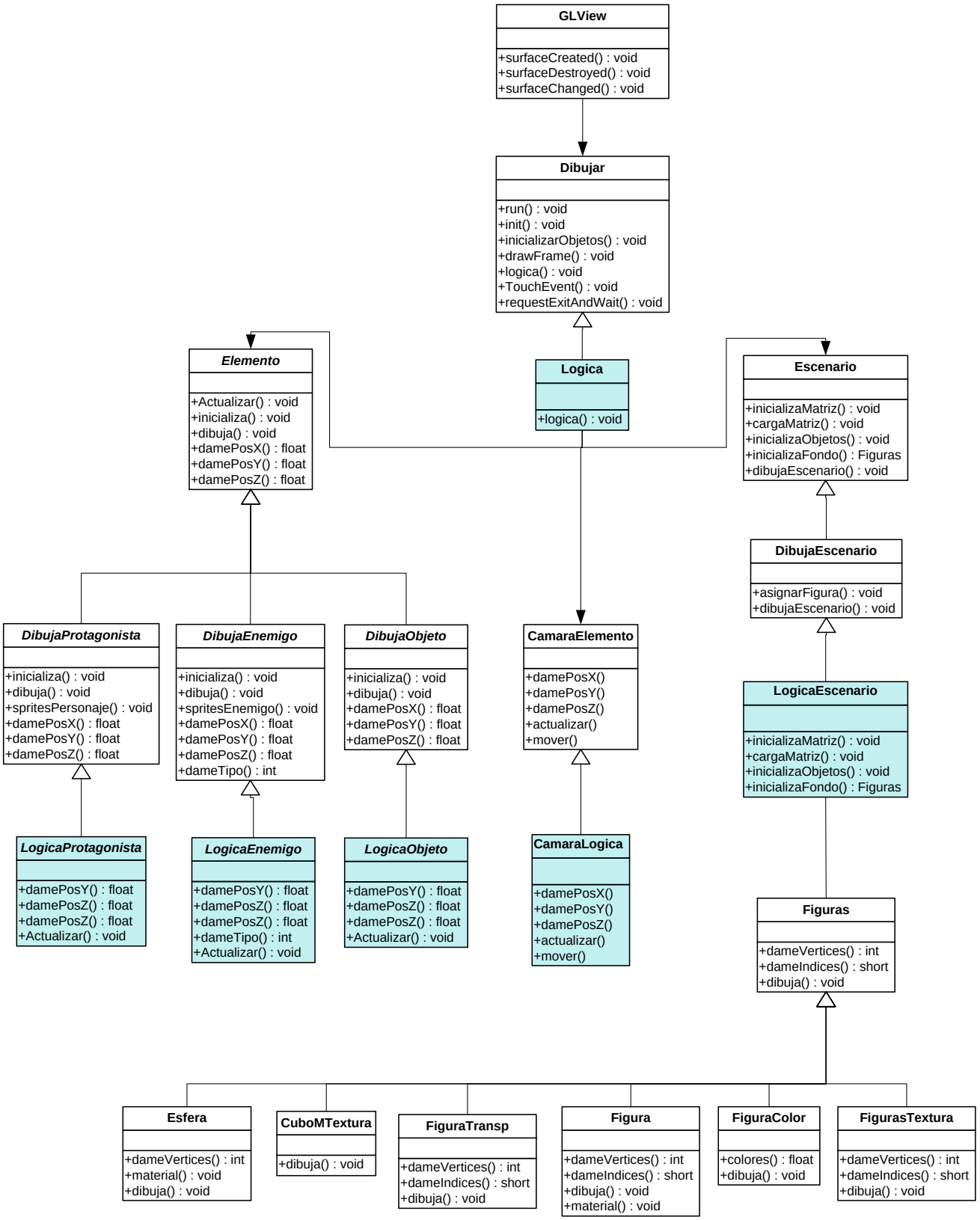


Figura 45: Diagrama de clases

4.1.2.- Definición de las clases

En el diagrama de clases que se muestra en la figura 45, no aparecen todas las clases que forman el sistema por motivos de espacio. Dichas clases se explicarán más detalladamente en la parte de implementación e irán acompañadas del diagrama de clases correspondiente.

- **GLView**

  Esta clase hereda de la clase SurfaceView e implementa SurfaceHolder. Callback. La clase SurfaceView es el punto de partida para la construcción de juegos que necesitan un nivel bastante alto de potencia a la hora de dibujar gráficos 2D o 3D.

  Para hacer uso del objeto Surface, es necesario hacerlo a través de un contenedor o SurfaceHolder e indicar posteriormente que dicho contenedor va a recibir las llamadas del SurfaceHolder. Callback. Estas tareas se realizan en el constructor de dicha clase y en el método cambia.

  Además de dicho método, se implementan tres métodos más, propios del interfaz SurfaceHolder. Callback:

  public void surfaceCreated (SurfaceHolder holder)

  Inicializa la superficie donde se va a dibujar el videojuego de plataformas.

  public void surfaceDestroyed (SurfaceHolder holder)

  Detiene el hilo principal y libera los recursos de la aplicación.

  public void surfaceChanged (SurfaceHolder holder, int format, int w, int h)

  Se invoca al cambiar el dispositivo de landascape a portraite. En este caso no se implementa debido a que la aplicación siempre será en un sentido de orientación.

- **Dibujar**

  public void run ()

  Este método se encarga de manejar todo el hilo de ejecución del videojuego. En él se inicializan todos los objetos necesarios para el correcto funcionamiento del mismo, y se ejecuta un bucle, que se encarga de llamar a los métodos lógica y dibujar, y de controlar el número de frames por segundo que la aplicación tarda en realizar estas dos tareas, hasta que el usuario ordena parar el videojuego, es decir, configura la variable muerto a true.

  public void init (GL10 gl)

  Este método se encarga de inicializar todos los elementos que forman el videojuego; el escenario, el protagonista, los objetos que obstaculizan al jugador alcanzar el final del nivel, el fondo, la cámara y los enemigos.

  public void drawFrame (GL10 gl)

  Este método se encarga de:

  - Establecer unas dimensiones máximas y mínimas de cada eje (x, y, z) de la pantalla.
  - Configurar la posición, el lugar al que apunta, la orientación, y el desplazamiento de la cámara, para que esta enfoque al protagonista en todo momento.
  - Dibujar todos los elementos que forman el videojuego.

  public abstract void logica ();

  Método abstracto que maneja toda la lógica del juego y será implementado por el usuario de la aplicación más adelante.

- **Logica**

  El nombre de esta clase es opcional, pero deberá heredar de Dibujar e implementar en su interior el método logica (), que se encargará de manejar las colisiones, el movimiento y las actualizaciones de los personajes que forman el juego, y el movimiento de la cámara.

- **Iluminacion**

  Clase utilizada para configurar las propiedades de iluminación de la escena.

- **Elemento**

  Clase abstracta formada por las variables posX, posY y posZ, que indican la posición de cada personaje del videojuego (protagonista, enemigos u objetos móviles) y por seis métodos abstractos que serán posteriormente implementados en las clases DibujaProtagonista, DibujaEnemigo, DibujaObjeto, LogicaProtagonista, LogicaEnemigo y LogicaObjeto.

- **DibujaProtagonista**

  Clase abstracta que hereda de Elemento, y está formada por 3 métodos que se encargan de inicializar el objeto protagonista, de dibujarlo y de actualizar los sprites del mismo a medida que este se desplaza por el escenario.

- **DibujaEnemigo**

  Clase abstracta que hereda de Elemento y está formada por 4 métodos, 1 de ellos abstracto que implementará el usuario de la librería. Esta clase se encarga de dibujar los enemigos y de actualizar los sprites de los mismos a medida que estos se desplazan por el escenario.

- **DibujaObjeto**

  Clase abstracta que hereda de Elemento, y está formada por dos métodos que se encargan de inicializar el objeto y de dibujarlo.

- **LogicaProtagonista**

  El nombre de esta clase es opcional, pero debe heredar de DibujaProtagonista e implementar los métodos damePosX (), damePosY (), damePosZ () y Actualizar (). De manera que los tres primeros devuelvan la posición del protagonista en cada eje de coordenadas (x, y, z) respectivamente, mediante un float, y el último se encargue de actualizar la posición del mismo.

- **LogicaEnemigo**

  El nombre de esta clase es opcional, pero debe heredar de DibujaEnemigo e implementar los métodos damePosX (), damePosY (), damePosZ (), dameTipo () y Actualizar (). Los tres primeros se encargarán de devolver la posición del enemigo en cada eje de coordenadas (x, y, z) respectivamente, mediante un float, el cuarto método de devolver un entero (int) con el tipo de enemigo a utilizar (un número comprendido del uno al tres) y el último de actualizar la posición del mismo.

- **LogicaObjeto**

  El nombre de esta clase es opcional, pero debe heredar de DibujaObjeto e implementar los métodos damePosX (), damePosY (), damePosZ () y Actualizar (). Los tres primeros se encargarán de devolver la posición del objeto en cada eje de coordenadas (x, y, z) respectivamente, mediante un float, y el último de actualizar la posición del mismo.

- **CamaraElemento**

  Clase abstracta formada por los métodos abstractos Actualizar (float X, float Y, float Z), Mover (), damePosX (), damePosY () y damePosZ ().

- **CamaraLogica**

  El nombre de esta clase es opcional, pero debe heredar de CamaraElemento e implementar los métodos damePosX (), damePosY (), damePosZ (), Actualizar (float X, float Y, float Z) y Mover (). Los tres primeros se encargarán de devolver la posición de la cámara mediante un float, el cuarto método se encargará de actualizar la posición de la cámara y el último de establecer los límites de giro del dispositivo y de actualizar sus valores.

- **Escenario**

  Clase abstracta formada por cinco métodos abstractos y dos variables; la primera de ellas es una matriz tridimensional denominada mundo que representa el escenario de juego, y la segunda es una variable que indica el radio de dibujado del sistema.

- **DibujaEscenario**

  Clase abstracta que hereda de Escenario e implementa los métodos asignarFigura () y dibujaEscenario (); el primero de ellos, recibe como parámetro una objeto de la clase Figuras y se encarga de introducir el mismo en un vector llamado figuras, creado en esa misma clase y el segundo, se encarga de dibujar el escenario siguiendo una serie de normas que se explicarán más adelante.

- **LogicaEscenario**

  El nombre de esta clase es opcional, pero debe heredar de DibujaEscenario e implementar los métodos inicializaMatriz (), inicializaObjetos (), inicializaFondo () y cargaMatriz (); el primero de ellos se encarga de inicializar la matriz mundo, el segundo de introducir todas las figuras que van a formar el escenario en el vector figuras creado en la clase DibujaEscenario, el tercero se encarga de devolver la figura que va a representar el fondo del nivel y el cuarto de rellenar la matriz mundo con números enteros.

  El usuario puede crear tantas clases como niveles o escenarios haya en el videojuego.

- **Figuras**

  Clase abstracta formada por tres métodos denominados dameVertices (), dameIndices () y dibuja (). Esta clase permite al usuario hacer uso de distintas figuras geométricas sencillas como triángulos, cuadrados, cubos, pirámides y esferas y añadir a las mismas distintas propiedades como la configuración de los materiales que las forman, el color o la texturización de sus caras.

- **Figura**

  Clase abstracta que hereda de la clase Figuras y está formada por dos métodos denominados material () y dibuja (). Esta clase permite al usuario hacer uso de distintas figuras geométricas y configurar los materiales que las forman.

- **FiguraColor**

  Clase abstracta que hereda de la clase Figuras y está formada por dos métodos denominados colores () y dibuja (). Esta clase permite al usuario hacer uso de distintas figuras geométricas y configurar el color de las mismas.

- **FiguraTextura**

  Clase abstracta que hereda de la clase Figuras y permite al usuario hacer uso de distintas figuras geométricas y mapear sobre la superficie de las mismas diversas texturas.

- **FiguraTransp**

  Clase abstracta que hereda de la clase Figuras y permite al usuario hacer uso de distintas figuras geométricas y mapear sobre la superficie de las mismas diversas texturas aplicando a las mismas un efecto de transparencia.

4.2.- Detalles de implementación

En primer lugar se comenzará desarrollando los detalles de implementación relacionados con la construcción de figuras, con distintos tipos de configuraciones como la iluminación, color y textura. Una vez desarrollada la construcción de todas ellas, se detallará el uso de las mismas para la creación de todos los elementos del videojuego.

4.2.1.- Introducción

Antes de comenzar a dibujar en OpenGL ES 1.0 será necesario conocer algunos aspectos fundamentales [65]:

1. En OpenGL ES 1.0 se dibuja sobre un sistema ortonormal de tres coordenadas x, y, z, donde la posición (0, 0 ,0) hace referencia al centro de este sistema.

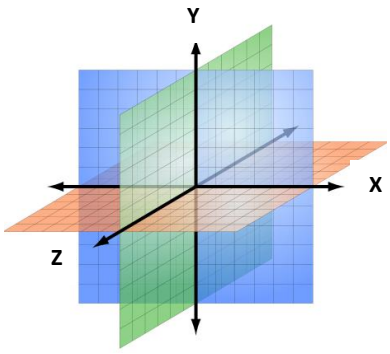


Figura 46: Sistema ortonormal

2. El sistema se podrá:

   - Desplazar: en cuyo caso el centro de referencia del sistema no corresponderá con la posición (0, 0, 0) sino con la posición indicada en el desplazamiento.

   - Rotar respecto a uno de sus ejes: en cuyo caso se indicará el ángulo de rotación y el eje sobre el cual se desea rotar.

   - Escalar: de tal manera que los objetos en él situados, aumenten o disminuyan su tamaño.

3. OpenGL ES 1.0 [71] provee al usuario interfaces de coma fija y coma flotante para todos sus métodos y variables. Normalmente, es preferible usar aritméticas en punto fijo debido a la carencia de unidad de coma flotante o FPU de algunos procesadores integrados, o a la limitada capacidad de cómputo de los mismos. Los métodos en coma fija terminan con la letra x, y los de coma flotante con la letra f, y las variables se definen mediante los tipos int para coma fija, y float para flotante.

4. En OpenGL ES 1.0 los objetos se definen a partir de un conjunto ordenado de vértices. El vértice es la unidad básica de dibujado y representa mediante 2, 3 o incluso 4 coordenadas, una posición en el espacio. Si se hace uso de 2 coordenadas se dibujará sobre un sistema bidimensional (x, y), donde OpenGL asignará de manera automática el valor 0 para la coordenada de profundidad z y el valor 1 para w. Si se utilizan 3 coordenadas (x, y, z) se dibujará sobre un sistema tridimensional, y en caso de usar 4 (x, y, z, w) sobre un sistema euclídeo donde el valor del punto en el espacio se corresponderá con la posición (x/w, y/w, z/w).

5. En OpenGL ES 1.0 una cara se forma mediante la unión ordenada de tres vértices, de tal manera que formen un triángulo. El orden de unión de los mismos es importante, pues a partir de él, OpenGL determina que caras son interiores y cuales exteriores, y por tanto visibles. Este sentido u orden de unión se puede especificar mediante el método:

   ```
   gl.glFrontFace (sentido);
   ```

   *Donde **sentido** toma el valor GL10.GL_CCW para la unión de los vértices en sentido antihorario y GL10.GL_CW para la unión de los vértices en sentido horario.*

   Además para facilitar la unión el usuario puede hacer uso de distintos tipos de primitivas o unidades básicas de dibujado. A continuación se muestra una tabla con todas ellas [67]:

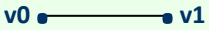
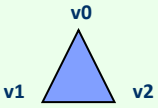
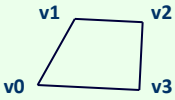
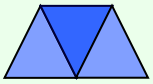
Valor	Primitiva	Dibujo
GL10.GL_POINTS	Puntos individuales	
GL10.GL_LINES	Línea: unión de 2 vértices	
GL10.GL_TRIANGLES	Triángulo: unión de 3 vértices	
GL10.GL_LINE_STRIP	Series de líneas conectadas	
GL10.GL_LINE_LOOP	Series de líneas conectadas unidas en un vértice	
GL10.GL_TRIANGLE_STRIP	Series de triángulos	
GL10.GL_TRIANGLE_FAN	Abanico de triángulos	

Tabla 1: Primitivas de OpenGL ES 1.0

4.2.2- Construcción de figuras.

El motor gráfico diseñado permite al usuario hacer uso de distintas figuras geométricas sencillas como triángulos, cuadrados, cubos y pirámides, que le ayudarán a desarrollar un videojuego de plataformas en 3D mediante el desplazamiento, la rotación o el escalado de las mismas y la iluminación, el color o la texturización de sus caras. Por ser dichas figuras el punto de partida en el diseño del videojuego, se comenzará con una explicación de los pasos más comunes, para la construcción de cualquiera de ellas.

1. Definir los vértices

   En OpenGL ES todas las figuras se definen a partir de la unión ordenada de vértices. Por lo tanto, el primer paso para la construcción de cualquiera de ellas, será definir un array de números enteros o reales de n vértices con dos, tres o cuatro coordenadas.

2. Almacenar de manera temporal la información de los vértices en un buffer de datos de la siguiente manera:

   - Crear un buffer de bytes de tamaño apropiado. Para pasar toda la información contenida en el array de vértices a un buffer de bytes, es necesario conocer el número de bytes que almacena cada tipo de datos del array. El array de vértices podrá estar formado de números enteros o de números reales, y ambos tipos reservan 4 bytes. Por ello el tamaño del buffer será igual a la longitud del array *4.
   - Conservar en el buffer, el orden original de los bytes.
   - Pasar dicho buffer de bytes al mismo tipo de datos que el array de vértices.
   - Añadir la información contenida en el array de vértices.
   - Colocar el puntero en la primera posición del buffer para comenzar a trabajar con él.

   A continuación se muestra un ejemplo del código utilizado:

```
ByteBuffer vbb = ByteBuffer.allocateDirect (vertices.length * 4);
vbb.order (ByteOrder.nativeOrder ());
vertexBuffer = vbb.asFloatBuffer ();
vertexBuffer.put (vertices);
vertexBuffer.position (0);
```

Código 1: Almacenar información de los vértices en un buffer de datos

3. Definir los índices

   Una vez definidos los vértices que forman la figura, será necesario unir los mismos. Mediante la construcción de un array de índices, el usuario podrá especificar el orden de unión de dichos vértices.

4. Almacenar de manera temporal la información de los índices en un buffer de la siguiente manera:

   - Crear un nuevo buffer de bytes cuyo tamaño sea la longitud del array de índices multiplicado por el número de bytes reservados por el tipo de datos de dicho array.
   - Conservar en el buffer, el orden original de los bytes.
   - Pasar dicho buffer de bytes al mismo tipo de datos que el array de índices.
   - Añadir la información contenida en el array de índices.
   - Colocar el puntero en la primera posición del buffer.

   A continuación se muestra un ejemplo del código utilizado [66]:

```
indexBuffer = ByteBuffer.allocateDirect (indices.length);
indexBuffer.put (indices);
indexBuffer.position (0);
```

Código 2: Almacenar información de los índices en un buffer de datos

5. Dibujar la figura

   Una vez obtenida la información necesaria, se hará uso de la misma para dibujar la figura elegida. Hay varias formas de dibujar en OpenGL ES 1.0:

   - Directamente desde el array de vértices, mediante la sentencia:

```
gl.glDrawArrays (mode, first, count);
```

     *Donde la variable **mode** indica el tipo de primitiva, la variable **first** el primer elemento del array desde el cual se comienza a dibujar la figura, y la variable **count** informa acerca del número de coordenadas que tendrá cada vértice.*

- Desde el array de índices, mediante la sentencia:

```
gl.glDrawElements (mode, count, type, indices);
```

*Donde la variable **mode** indica el tipo de primitiva, **count** el número de coordenadas de cada índice, **type** el tipo de datos (GL_UNSIGNED_BYTE o GL_UNSIGNED SHORT) e **indices** el nombre del buffer donde se almacenó de manera temporal la información de los índices.*

En este motor gráfico se siguen todos estos pasos para la construcción de la mayoría de las figuras. Por ello, se implementa la clase abstracta ***Figuras*** formadas por tres métodos abstractos denominados *dameVertices*, *dameIndices* y *dibujar*. Los métodos *dameVertices* y *dameIndices,* se implementan en los cuerpos de las clases de cada tipo de figura, ya que cada una de ellas, estará formada por un número distinto de vértices. A continuación se explica la implementación de estos métodos para un triángulo, un cuadrado, una pirámide y un cubo.

4.2.2.1- Construcción de un triángulo.

Para la construcción de un triángulo [70] se crea un método denominado *dameVertices*, al que se le pasa como parámetro tres valores enteros (*x, y, z*) y devuelve un array formado por 3 vértices con 3 coordenadas cada uno. Las coordenadas *x* e *y* serán seleccionadas por el usuario, de tal manera que la primera de ellas indicará el valor más grande y el valor más pequeño en el eje de las x, es decir, la anchura del triángulo y la segunda, el valor más grande y el más pequeño en el eje de las y, es decir, la altura del triángulo. La coordenada *z* tomará el valor de 0 por tratarse de una figura plana.

```
public int [] dameVertices (int x, int y, int z)
{
      int vertices [] =
      {
            0,  y, 0,   // v0. Arriba
           -x, -y, 0,   // v1. Abajo izquierda
            x, -y, 0    // v2. Abajo derecha
      };
      return vertices;
}
```

Código 3: Coordenadas vértices triángulo

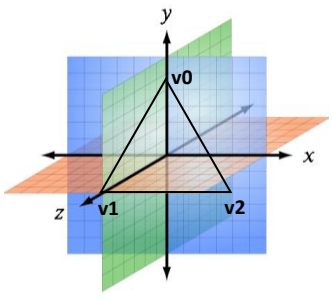


Figura 47: Vértices triángulo

Una vez creados los tres vértices que forman el triángulo, se unen los mismos para la formación de caras. Para ello, se implementa un método llamado *dameIndices*, que devuelve un array de bytes con 3 índices.

A continuación se muestra el código utilizado:

```
public short [] dameIndices ()
{
	short indices [] = {0, 1, 2};
	return indices;
}
```

Código 4: Índices triángulo

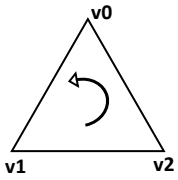


Figura 48: Índices triángulo

4.2.2.2.- Construcción de un cuadrado

Para la construcción de un cuadrado [70] se crea un método *dameVertices*, al que se le pasa como parámetro tres valores enteros (*x, y, z*) y devuelve un array formado por 4 vértices con 3 coordenadas cada uno.

A continuación se muestra el código utilizado:

```
public int [] dameVertices (int x, int y, int z)
{
	int [] vertices =
	{
		 x, -y, 0, // v0, Abajo derecha
		-x, -y, 0, // v1, Abajo izquierda
		 x,  y, 0, // v2, Arriba derecha
		-x,  y, 0, // v3, Arriba izquierda
	};
	return vertices;
}
```

Código 5: Coordenadas vértices cuadrado

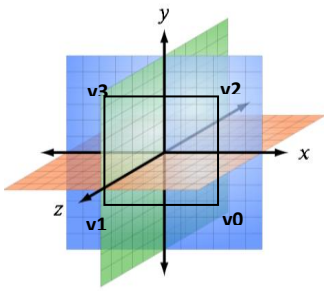


Figura 49: Vértices cuadrado

Una vez definidos los 4 vértices, se unen los mismos en grupos de tres en tres, de manera que se formen dos caras. Para ello, se implementa un método llamado *dameIndices*, que devuelve un array de bytes formado por 6 valores, los 3 primeros forman una cara y los 3 últimos otra.

A continuación se muestra el código utilizado:

```
public short [] dameIndices ()
{
	byte [] indices = {2, 0, 1, 2, 3, 1};
	return indices;
}
```

Código 6: Índices cuadrado

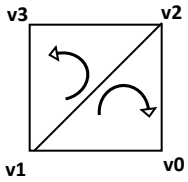


Figura 50: Índices cuadrado

4.2.2.3.- Construcción de una pirámide

Para la construcción de una pirámide, sin base, [70] se crea el método *dameVertices*, al que se le pasa como parámetro tres enteros (x, y, z) y devuelve un array de 12 vértices con 3 coordenadas cada uno. Los valores x, y, z serán seleccionados por el usuario, de tal manera que el valor x establecerá el máximo y el mínimo valor en el eje x, es decir, el ancho de la pirámide, el valor y, indicará el máximo y el mínimo valor en el eje y, es decir, el alto de la pirámide y el valor z el máximo y el mínimo valor en el eje z, es decir, el largo de la pirámide.

A continuación se muestra el código utilizado:

```
public int [] dameVertices (int x, int y, int z)
{
	int vertices [] =
	{
		 0,  y,  0,  //v0. Superior (frontal)
		-x, -y,  z,  //v1. Izquierda (frontal)
		 x, -y,  z,  //v2. Derecha (frontal)
		 0,  y,  0,  //v3. Superior (derecha)
		 x, -y,  z,  //v4. Izquierda (derecha)
		 x, -y, -z,  //v5. Derecha (derecha)
		 0,  y,  0,  //v6. Superior (trasera)
		 x, -y, -z,  //v7. Izquierda (trasera)
		-x, -y, -z,  //v8. Derecha (trasera)
		 0,  y,  0,  //v9. Superior (izq.)
		-x, -y, -z,  //v10.Izquierda (izq.)
		-x, -y,  z   //v11.Derecha (izquierda)
	};
	return vertices;
}
```

Código 7: Coordenadas vértices pirámide

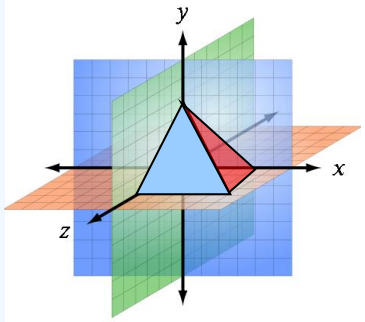


Figura 51: Vértices pirámide

Una vez definidos los 12 vértices, se procede a la unión de los mismos para la formación de caras. Para ello, se implementa el método llamado *dameIndices*, que devuelve un array de bytes formado por 4 grupos de 3 vértices cada uno. Para formar una pirámide, sin base, será necesaria la unión de 4 caras.

A continuación se muestra el código utilizado:

```
public short [] dameIndices ()
{
	short indices [] =
	{
		0,  1,  2, // cara frontal
		3,  4,  5, // cara derecha
		6,  7,  8, // cara trasera
		9, 10, 11, // cara izq.
	};
	return indices;
}
```

Código 8: Índices pirámide

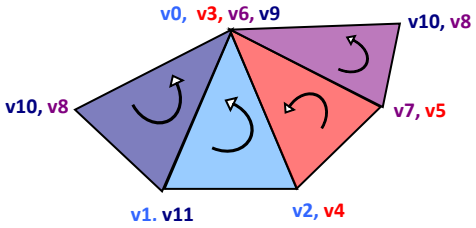


Figura 52: Índices pirámide

4.2.2.4.- Construcción de un cubo

Para la construcción de un cubo [70] también se crea un método *dameVertices*, al que se le pasa como parámetro tres enteros (x, y, z) y devuelve un array de enteros de 24 vértices con 3 coordenadas (x, y, z) cada uno.

A continuación se muestra el código utilizado:

```
public int [] dameVertices (int x, int y, int z)
{
	int [] vertices =
	{
		-x, -y, z, //v0 Abajo izq. (cara frontal)
		 x, -y, z, //v1 Abajo dcha. (cara frontal)
		-x,  y, z, //v2 Arriba izq. (cara frontal)
		 x,  y, z, //v3 Arriba derecha (cara frontal)

		 x, -y,  z, //v4 Abajo izq. (cara dcha.)
		 x, -y, -z, //v5 Abajo dcha. (cara dcha.)
		 x,  y,  z, //v6 Arriba izq. (cara dcha.)
		 x,  y, -z, //v7 Arriba dcha. (cara dcha.)
```

```
                x, -y, -z, //v8  Abajo izq. (cara trasera)
               -x, -y, -z, //v9  Abajo dcha. (cara trasera)
                x,  y, -z, //v10 Arriba izq. (cara trasera)
               -x,  y, -z, //v11 Arriba dcha. (cara trasera)

               -x, -y, -z, //v12 Abajo izq. (cara izq.)
               -x, -y,  z, //v13 Abajo dcha. (cara izq.)
               -x,  y, -z, //v14 Arriba izq. (cara izq.)
               -x,  y,  z, //v15 Arriba dcha. (cara izq.)

               -x, -y, -z, //v16 Abajo izq. (cara inferior)
                x, -y, -z, //v17 Abajo derecha (cara inferior)
               -x, -y,  z, //v18 Arriba izq. (cara inferior)
                x, -y,  z, //v19 Arriba derecha (cara inferior)

               -x,  y,  z, //v20 Abajo izq. (cara superior)
                x,  y,  z, //v21 Abajo derecha (cara superior)
               -x,  y, -z, //v22 Arriba izq. (cara superior)
                x,  y, -z, //v23 Arriba derecha (cara superior)

        };
        return vertices;
}
```

Código 9: Coordenadas vértices cubo

Una vez definidos los 24 vértices, se procede a la unión de los mismos para la formación de caras. Para ello, se implementa el método llamado *dameIndices*, que devuelve un array de bytes formado por 12 grupos de 3 vértices cada uno. Para formar una cubo será necesaria la unión de 12 caras, 2 por cada lado del cubo.

A continuación se muestra el código utilizado:

```
public short [] dameIndices ()
{
        short indices [] =
        {
                0,  1,   3,   0,  3,  2,
                4,  5,   7,   4,  7,  6,
                8,  9,  11,   8, 11, 10,
                12, 13, 15,  12, 15, 14,
                16, 17, 19,  16, 19, 18,
                20, 21, 23,  20, 23, 22,
        };
        return indices;
}
```

Código 10: Índices cubo

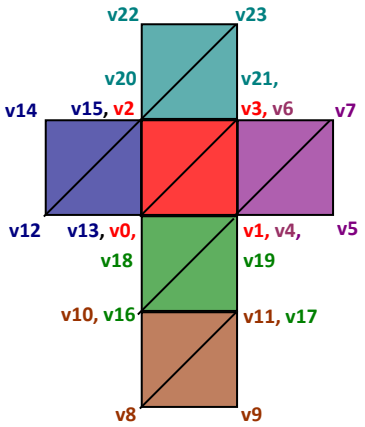


Figura 53: Índices cubo

4.2.2.1.- Otras figuras

Para la construcción de una esfera no es necesario implementar el método *dameIndices ()* ni almacenar la información del array de índices de manera temporal en un buffer de datos. Por este motivo, se tratará esta figura de manera independiente. La clase ***Esfera*** hereda de la clase ***Figuras*** y está formada por un constructor, que se encarga de crear el buffer que contendrá información acerca de los puntos que forman la misma, y de dos métodos llamados *dameVertices* y *dibuja*.

El método *dameVertices* se encarga de calcular las coordenadas cartesianas (x, y, z) en el sistema de coordenadas esféricas (r, ϕ, θ), de introducir las mismas en el buffer creado en el constructor, y de devolver el número total de puntos utilizados para la formación de la esfera [36] y el método *dibuja* se encarga de dibujar todos los puntos de la esfera.

Para el cálculo de las coordenadas cartesianas se hace uso de la siguiente fórmula:

$$\begin{cases} x &= r\sin\theta\ \cos\phi \\ y &= r\sin\theta\sin\ \phi \\ z &= r\cos\theta \end{cases} \qquad \text{con } -\frac{\pi}{2} < \theta \leq \frac{\pi}{2}, \quad \text{y} \quad 0 < \phi \leq \pi$$

A continuación se muestra el código necesario para la creación del método *dameVertices*:

```
private int dameVertices ()
{
	int points  = 0;
	for (double theta= 0; theta<= Math.PI; theta=theta+0.03f)
	{
		for (double phi = 0.0; phi<=(Math.PI*2); phi=phi+0.03f)
		{
			verticesBuffer.put ((float) (mRaduis * Math.sin
			(theta) * Math.cos (phi)));
			verticesBuffer.put ((float) (mRaduis * Math.sin
			(theta) * Math.sin (phi)));
			verticesBuffer.put ((float) (mRaduis * Math.cos
			(theta)));
			points ++;
		}
	}
	verticesBuffer.position (0);
	return points;
}
```

Código 11: Puntos de una esfera

Una vez construidas las figuras se aplican sobre ellas diferentes transformaciones. A continuación se explican de manera detallada cada una de ellas.

4.2.3.- Iluminación de figuras.

Antes de comenzar con la iluminación de las figuras es necesario iluminar la escena. Hay dos modelos distintos de iluminación [34]:

- *La iluminación global*: se calcula la iluminación de la escena teniendo en cuenta cada uno de los polígonos existentes en la misma y su interacción con la luz.
- *La iluminación local*: hace un cálculo, en tiempo real, de la iluminación teniendo en cuenta exclusivamente la luz directamente reflejada por el objeto.

OpenGL ES 1.0 utiliza el segundo modelo, de modo que cada polígono se sombrea de manera independiente.

Si el usuario desea habilitar el uso de luces y configurar la iluminación de la escena y de las figuras, deberá crear un objeto de la clase ***Iluminación***, y llamar al método *inicializa* de dicha clase. A continuación se detallan los pasos seguidos para su implementación:

1. **Habilitar el uso de luces**

   Para hacer uso de la iluminación en primer lugar es necesario habilitar esta propiedad mediante la sentencia [23]:

   ```
   gl.glEnable (GL10.GL_LIGHTING);
   ```

   Habilitar las fuentes de luz indicadas por el usuario mediante la sentencia:

   ```
   gl.glEnable (int fuente);
   ```

   *Donde **fuente** puede tomar los valores GL10.GL_LIGHT0, GL10.GL_LIGHT1, GL10.GL_LIGHT2, GL10.GL_LIGHT3, GL10.GL_LIGHT4, GL10.GL_LIGHT5, GL10.GL_LIGHT6 y GL10.GL_LIGHT7.*

   El usuario indicará al sistema el número de la fuente de luz que quiere configurar, pasándole como primer parámetro al método *inicializa* un entero del 0 al 7.

2. Especificar las luces

Para configurar la luz global que interactúa con la superficie de un objeto es necesaria la utilización de tres parámetros [72]:

- *Luz ambiental*: Los rayos que inciden sobre el objeto provienen de todas las direcciones y no generan ningún tipo de sombreado sobre la superficie del mismo. La luz ambiental, es originada por los sucesivos rebotes de luces, sobre las superficies de los objetos que forman la escena.

- *Luz difusa*: En este caso, los rayos provienen de una fuente de luz puntal e identificable. Al incidir la luz sobre el objeto, parte de la misma se reflejará en una sola dirección. La iluminación del objeto dependerá de su forma y de la relación de posiciones cámara-objeto-fuente de luz.

- *Componente de brillo o luz especular*: Los rayos inciden de manera paralela al objeto. Si el objeto es brillante, producirá zonas donde la luz se refleja más intensamente, si es mate, la luz se reflejará en todas direcciones.

Estos parámetros se configuran mediante cuatro variables, las tres primeras indican los valores RGB, respectivamente y la última, el tipo de luz a utilizar. OpenGL ES 1.0 utiliza tres tipos de luces:

- *La luz direccional:* fuente de luz muy lejana cuyos rayos inciden en el objeto de manera paralela. Se indicará a OpenGL ES el uso de este tipo de luz mediante la configuración del cuarto parámetro con w=0.0.

- *Luz puntual:* luz proveniente de un punto concreto que ilumina en todas direcciones. El uso de esta luz se indicará configurando la cuarta coordenada con w=1.0.

- *Luz focal:* luz emitida por un foco que ilumina en una única dirección y con un determinado ángulo. Al tratarse de un tipo de luz puntual, se configurará el cuarto parámetro con w=1.0. A parte, se especificará la posición del foco de luz, GL10.GL_SPOT_DIRECCION, su ángulo, GL10.GL_SPOT_CUTOFF, y su atenuación GL10.GL_SPOT_EXPONENT.

Además también es posible configurar las posiciones de la luz mediante cuatro parámetros, los tres primeros indican la posición de la luz mediante las coordenadas x, y, z y el último, el tipo de luz utilizado.

El usuario pasará como parámetros al método *inicializa* cuatro matrices de tipo float, la primera de ellas configurará los parámetros de la luz ambiental, la segunda los de la luz difusa, la tercera los de la luz especular y la última la posición de la luz. A continuación se muestra un ejemplo de cómo el usuario deberá configurar los parámetros de la luz y llamar al método *inicializa*:

```
float ambiente [] = {0.75f, 0.75f, 0.75f, 1.0f};
float difusa [] = {1.0f, 1.0f, 1.0f, 1.0f};
float especular [] = {1.0f, 1.0f, 1.0f, 1.0f};
float posicion [] = {0.0f, 0.0f, 1.0f, 0.0f};

Iluminacion iluminacion = new Iluminacion ();
ilu.inicializa (1, gl, ambiente, difuso, especular, posicion);
```

Código 12: Configuración de los parámetros de luz

Una vez realizado este paso, el usuario podrá configurar las figuras que forman el videojuego para que las superficies de las mismas estén formadas por materiales con unas características concretas. Para ello, deberá hacer uso de la clase abstracta ***Figura*** y de las clases ***Cubo, Cuadrado, Piramide***, ***Triangulo*** y ***Esfera***.

La clase ***Figura*** hereda de la clase ***Figuras*** y está formada por un constructor que se encarga de almacenar de manera temporal la información de los arrays de vértices y de índices en un buffer de datos, y por los métodos *dibuja* y *material*; el primero de ellos se encarga de dibujar las figuras y el segundo, de establecer las propiedades de los materiales que las forman.

Las clases ***Cubo, Cuadrado, Piramide*** y ***Triangulo*** heredan de la clase ***Figura*** e implementan en su interior los métodos *dameVertices* y *dameIndices* conforme a lo explicado en el apartado 4.2.2.

La clase ***Esfera*** hereda de la clase ***Figuras***, y está formada por 4 métodos, *dameVertices*, *dameIndices, material* y *dibuja.*

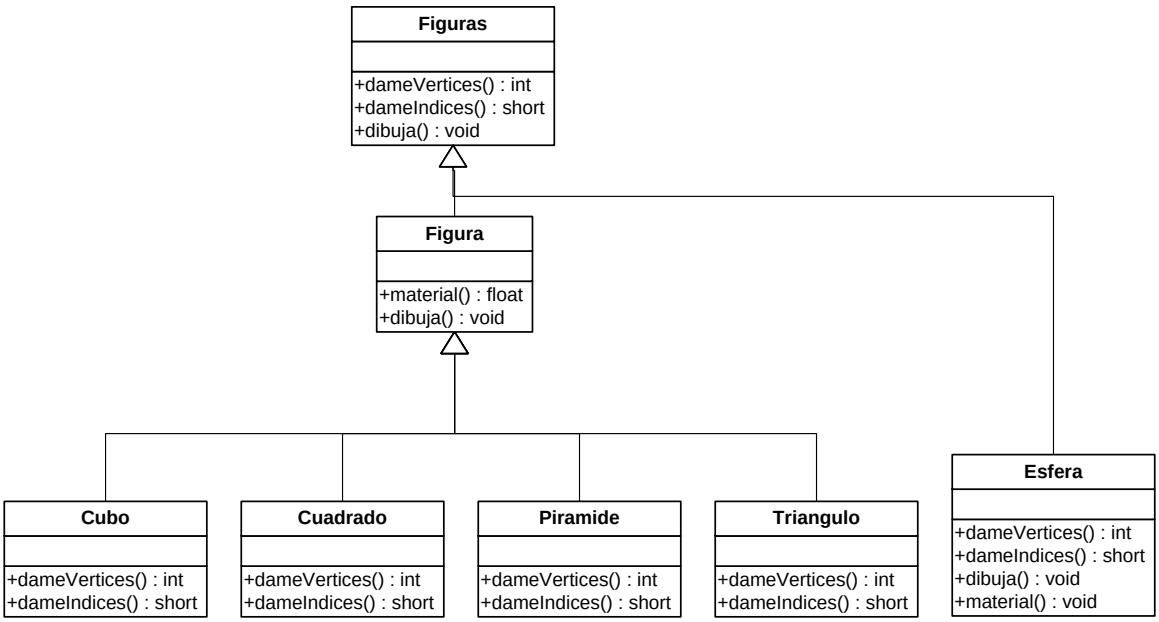


Figura 54: Diagrama de clases Figura

Para la implementación del método *material* se llevan a cabo los siguientes pasos:

1. Especificar el sombreado de las superficies

OpenGL ES 1.0 usa un modelo simplificado para simular el sombreado de superficies en el mundo real. Hay tres tipos básicos de modelos o sombreados [23].

- *Sombreado plano*: evalúa el color del primer vértice y asigna el mismo a todos los puntos del polígono. Esta técnica es bastante sencilla y los cálculos requeridos para llevarla a cabo son mínimos, sin embargo resta realismo a la escena. Para hacer uso de la misma se utiliza la sentencia:

```
gl.glShadeModel (GL10.GL_FLAT);
```

- *Sombreado suave o GOURAUD*: evalúa el color de los vértices del polígono e implementa una interpolación de colores para el resto de los puntos. Esta técnica genera escenas más realistas con una calidad notable, sin embargo no consigue simular las curvas.

```
gl.glShadeModel (GL10.GL_SMOOTH);
```

- *Sombreado PHONG*: Esta técnica utiliza una interpolación bilineal para calcular la normal de cada uno de los puntos del polígono. A pesar de que proporciona una resultados muy buenos y próximos a la realidad, es la más costosa y OpenGL no la implementa de manera directa.

El usuario indicará al sistema el tipo de sombreado que desea utilizar pasándole como primer parámetro al método *material* un entero, si el entero toma el valor 0 se utilizará el sombreado plano, si toma cualquier otro valor, se utilizará el sombreado suave.

2. Especificar las normales de las superficies

La iluminación en OpenGL se calcula mediante los focos de luz activos en ese momento y la incidencia de estas luces sobre los vértices de la figura. Para saber el ángulo de incidencia de la luz sobre estos vértices es necesario calcular un vector perpendicular a la cara que se esté dibujando en ese momento, o dicho de otro modo, calcular la normal del vértice. Dicho vector puede ser modificado para conseguir distintos efectos sobre la superficie de la figura y su longitud debe de ser 1.

3. Especificar las propiedades de los materiales

Una vez que se habilita el uso de la iluminación, los efectos de color son ignorados, por ese motivo si se desea dar color o brillo a un objeto será necesario definir los materiales a partir de los cuales están formados.

Los parámetros GL10.GL_AMBIENT y GL10.GL_DIFFUSE, especificarán el color del material, los parámetros GL10.GL_SPECULAR y GL10.GL_SHININESS se utilizará para configurar el brillo del mismo y GL10.GL_EMISSION indicará si es emisor de luz o no.

Una vez creado el material OpenGL ES proporciona la posibilidad de aplicarlo en las caras frontales de la figura a través del comando GL10.GL_FRONT o en todas las caras, a través de GL10.GL_FRONT_AND_BACK. El motor gráfico utilizará la segunda opción por defecto.

El usuario pasará como parámetros al método *material* cuatro matrices de tipo float para configurar los parámetros ambiente, difuso, especular y emisivo [19] del material, y una variable de tipo float para establecer el brillo del mismo.

En resumen, para crear y configurar cualquiera de estas figuras, el usuario deberá seguir los siguientes pasos:

1. Iluminar la escena.
2. Generar un objeto de la clase ***Figura*** y pasarle al constructor de dicha clase 5 parámetros, el primero de ellos corresponderá con una instancia del objeto de la clase GL10, el segundo indicará al sistema la vista sobre la cual dibujar, y los tres últimos indicarán las dimensiones de la figura en números enteros, de tal manera que un 1 en números reales corresponderá con un 2^{16} en números enteros.

   Generar un objeto de la clase ***Figuras*** y pasarle un constructor con el tamaño del radio de la esfera deseado en números reales.
3. Configurar los parámetros del material que la forman.
4. Llamar al método *material*.

A continuación se muestra un ejemplo del código necesario para configurar una pirámide formada por cobre y de una esfera formada por ese mismo material:

```
float ambiente [] = new float [] {0.33f, 0.26f, 0.23f, 1.0f};
float difuso [] = new float [] {0.5f, 0.11f, 0.0f, 1.0f};
float especular [] = new float [] {0.95f, 0.73f, 0.16f, 1.0f};
float emisor [] = new float [] {0.0f, 0.0f, 0.0f, 1.0f};
float brillo =93.0f;

piramide.material (0, gl, ambiente, difusa, especular, emisor, brillo);

piramide =new Piramide (gl, view.getContext (),36384,62768,32000);

esfera.material (0, gl, ambiente, difusa, especular, emisor, brillo);

esfera =new Esfera (1.0f);
```

Código 13: Crear y configurar pirámide de cobre

4.2.4.- Figuras con color.

Si el usuario desea aplicar un color sobre la superficie de las figuras que forman el videojuego de plataformas en 3D, deberá hacer uso de la clase abstracta ***FiguraColor***, y de las clases ***CuboColor***, ***CuadradoColor***, ***PiramideColor*** y ***TrianguloColor***.

La clase ***FiguraColor*** hereda de la clase ***Figuras*** y está formada por un constructor que se encarga de almacenar de manera temporal la información de los arrays de vértices y de índices en un buffer de datos, y por los métodos *dibuja* y *colores*. Las clases ***CuboColor***, ***CuadradoColor***, ***PiramideColor*** y ***TrianguloColor***, heredan de la clase ***FiguraColor*** e implementan en su interior los métodos *dameVertices* y *dameIndices*, conforme a lo explicado en el apartado 4.2.2, y los métodos *dameColores* y *dibuja,* que se explicarán más adelante.

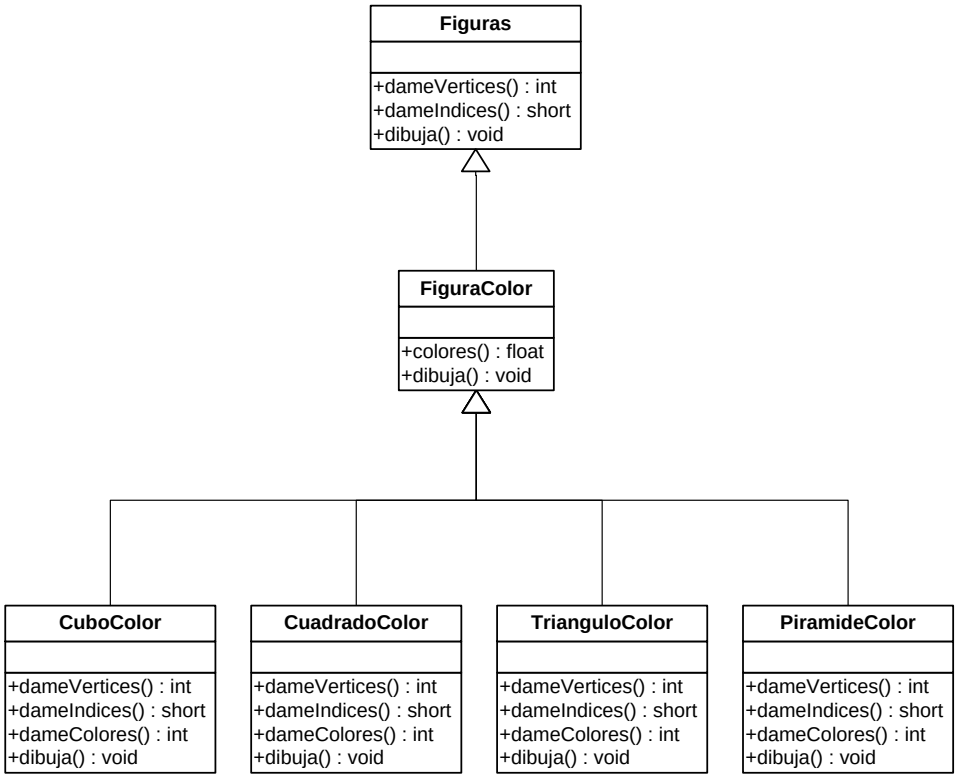


Figura 55: Diagrama de clases FiguraColor

Para implementar el método *colores* previamente fue necesario escoger el tipo de modelo de color a utilizar en el motor gráfico. OpenGL ES ofrece al usuario tres tipos distintos de modelos de color:

- *El modelo RGB* [13]: es un modelo aditivo a través del cual es posible generar el color de un píxel mediante la mezcla del rojo, verde y azul. Los colores en modo RGB se pueden representar por medio de un cubo situado sobre tres ejes de coordenadas que representan los colores anteriormente mencionados. De tal manera que cada punto del cubo representa un color, y las coordenadas de dichos puntos la cantidad de rojo, verde y azul de los mismos.

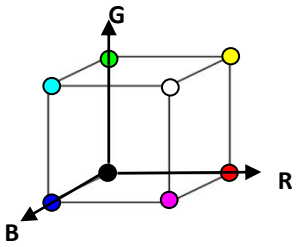


Figura 56: Modelo RGB

Para indicar a OpenGL en qué proporción se utiliza cada color, se asigna un valor a cada una de las componentes en el intervalo de [0, 1], para números reales, y de [0, 255], para números enteros. El valor mínimo indicará la ausencia de ese color y el máximo la saturación total del mismo.

La cantidad de colores a representar dependerá de la tarjeta de vídeo del sistema. Si la tarjeta permite un uso de 4 bits para representar el color de un píxel, el sistema podrá mostrar 16 colores, del mismo modo, si tiene 8 bits, se podrán mostrar 256 colores y así sucesivamente. Si al llamar a la función encargada de generar el color (gl.glColor) el sistema es incapaz de representar dicho color, OpenGL ES 1.0 se encargará de seleccionar en su lugar el color más próximo al escogido por el usuario.

- *El modelo RGBA*: Los tres primeros valores especifican un color por medio de las componentes rojo, verde y azul y el cuarto valor, indica la transparencia de ese color por medio del canal alpha.

- *El modo índices de color*: Los colores se encuentran ordenados de tal manera que cada color tiene un índice asociado. Por lo tanto, si se desea hacer uso de un color, se accederá a él a través de su índice.

El modelo de color utilizado para la implementación del método *colores* fue el modelo RGBA. Dicho método recibe como parámetro una cadena con el nombre de un color y devuelve un array de números reales con cuatro valores, el primero de ellos corresponde con la cantidad de rojo de dicho color, el segundo de ellos con la cantidad de verde, el tercero con la cantidad de azul y el cuarto con la transparencia del mismo.

Una vez establecido el modelo de color, se procederá a implementar los métodos *dameColores* y *dibuja,* de las clases ***TrianguloColor, CuboColor***, ***CuadradoColor*** y ***PiramideColor.*** El método *dameColores,* se encarga de devolver un array de números reales que representa el color de cada uno de los vértices de la figura, y el método *dibuja,* de dibujar cada vértice, unir las primitivas y colorearlas.

Para configurar y crear cualquiera de estas figuras, el usuario deberá crear un objeto de la clase ***FiguraColor***. El número de parámetros que le pasará al constructor dependerá del tipo de figura que desee dibujar. Si el usuario desea:

- Dibujar un triángulo a color: deberá pasar al constructor 8 parámetros, los 3 últimos indicarán el nombre del color de cada uno de los vértices (v0, v1, v2) del triángulo.

- Dibujar un cuadrado a color: deberá pasar al constructor 9 parámetros, los 4 últimos indicarán el nombre del color de cada uno de los vértices (v0, v1, v2, v3) del cuadrado.

- Dibujar una pirámide a color: el usuario deberá pasar al constructor 10 parámetros, los 5 últimos indicarán el nombre del color de cada uno de los vértices de la pirámide.

- Dibujar un cubo a color: el usuario deberá pasar al constructor 13 parámetros, los 8 últimos indicarán el nombre del color de cada uno de los vértices del cubo.

Los 5 primeros parámetros de todas ellas serán comunes (el primero de ellos corresponderá con una instancia del objeto de la clase GL10, el segundo con la vista sobre la cual dibujar, y los demás con las dimensiones de la figura).

Los colores que el motor gráfico ofrece y los nombres para usar los mismos son los siguientes:

Nombre	Código modelo ARGB	Color
rojo	{1.0f,0.0f,0.0f,1.0f}	
verde	{0.0f,1.0f,0.0f,1.0f}	
azul	{0.0f,0.0f,1.0f,1.0f}	
fucsia	{1.0f,0.0f,1.0f,1.0f}	
amarillo	{1.0f,1.0f,0.0f,1.0f}	
azulcielo	{0.0f,1.0f,1.0f,1.0f}	
blanco	{1.0f,1.0f,1.0f,1.0f}	
negro	{0.0f,0.0f,0.0f,1.0f}	
naranja	{1.0f,0.27f,0.0f,1.0f}	
morado	{0.62f,0.12f,0.94f,1.0f}	
marron	{0.54f,0.27f,0.07f,1.0f}	
gris	{0.5f,0.5f,0.5f,1.0f}	

Tabla 2: Colores predefinidos del motor gráfico

Una vez establecido el color de cada vértice de la figura, se deberá pintar el interior de la misma. OpenGL ES 1.0 ofrece al usuario dos modelos de colorear primitivas:

- *Modelo plano*: Si se desea colorear una línea mediante este modelo OpenGL asignará el color del último vértice de la primitiva al interior de toda la línea. Si se desea colorear el interior de una figura OpenGL asignará el color del primer vértice de la primitiva sobre el interior de la misma.

- *Modelo suavizado*: el interior de las líneas y los polígonos se colorea haciendo una interpolación o suavizado entre los colores asociados a los vértices. Este modelo es el que OpenGL utiliza por defecto y el que se utilizará en el motor gráfico.

A continuación se muestra un ejemplo del código necesario para crear y configurar un triángulo, un cuadrado, una pirámide y un cubo, con color:

```
piramideColor =new PiramideColor (gl, view.getContext (), 58000, 52000,
58000, "rojo", "rojo", "amarillo", "amarillo", "naranja");
cuboColor =new CuboColor (gl, view. getContext (), 36384, 62768, 32000,
"morado", "morado", "morado", "morado", "azul", "azul", "gris",
"gris");
trianguloColor =new TrianguloColor (gl, view. getContext (), 36384,
62768, 32000, "verde", "azul", "rojo");
cuadradoColor  =new  CuadradoColor  (gl,  view.  getContext  (),  58000,
52000, 58000, "fucsia", "gris", "azulcielo", "blanco");
```

Código 14: Dibujar figuras en color

Aparte de colorear las figuras, el usuario podrá pintar el fondo del videojuego de un color determinado. OpenGL ES almacena la información de color de cada píxel de la ventana en un buffer, por ello lo primero que el usuario deberá hacer será inicializar dicho buffer de color mediante la siguiente sentencia:

```
gl.glClear (GL10.GL_COLOR_BUFFER_BIT);
```

Una vez inicializado se guardará en dicho buffer la información de color especificada en el siguiente método y se establecerá dicho color en el fondo de la ventana en formato RGBA.

```
gl.glClearColor (float red, float green, float blue, float alpha);
```

*Donde **red, green, blue** y **alpha** corresponden con las cuatro componentes de color del modelo RGBA y sus valores estarán comprendidos entre 0 y 1 para números reales.*

4.2.5.- Figuras con texturas.

Si el usuario desea aplicar una imagen sobre las superficies de las figuras que forman el videojuego de plataformas en 3D, deberá hacer uso de la clase abstracta ***FiguraTextura***, y de las clases ***CuboTextura***, ***CuadradoTextura***, ***PiramideTextura*** y ***TrianguloTextura***.

La clase ***FiguraTextura*** hereda de la clase ***Figuras***, y hace una llamada en su constructor a una clase llamada ***Texturas***. Las clases ***CuboTextura***, ***CuadradoTextura***, ***PiramideTextura*** y ***TrianguloTextura***, heredan de la clase ***FiguraTextura*** e implementan en su interior los métodos *dameVertices*, *dameIndices* y *dibuja*, conforme a lo explicado en el apartado 4.2.2.

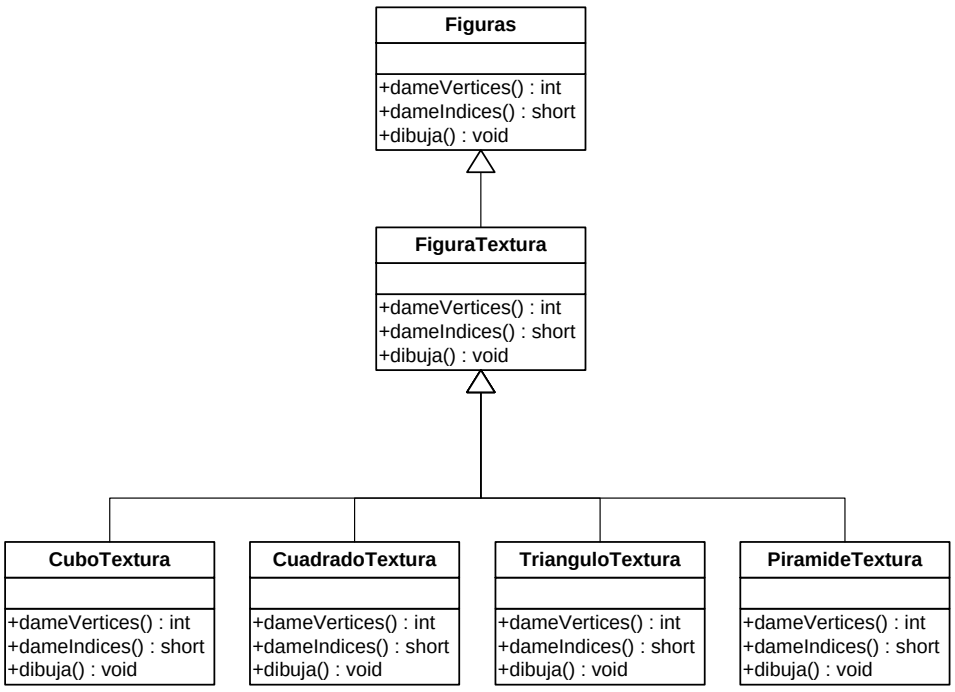


Figura 57: Diagrama de clases FiguraTextura

Se comenzará a explicar la clase ***Texturas***, por ser la encargada de extraer la información que representa a una imagen en un buffer de datos, mediante la implementación del método *extraer,* y de cargar las texturas requeridas en la memoria y configurar las propiedades de cada una de ellas, mediante el método *load*. Para implementar la clase ***Texturas*** previamente fue necesario escoger el tipo de textura adecuado.

Una textura, es un array de datos que representa una imagen. Dependiendo del tipo de textura que se utilice se almacenarán más o menos datos. OpenGL ES 1.0 ofrece al usuario tres tipos distintos [69]:

- **Las texturas unidimensionales**: se caracterizan por tener un solo píxel de alto o de ancho, y se suelen emplear para representar un gradiente de color sobre el objeto.

- **Las texturas bidimensionales**: son mapas de bits que guardan información de color de cada píxel de una imagen rectangular o cuadrada, y permiten la caracterización de personajes o la representación de un patrón a repetir como la superficie de un escenario.

- **Las texturas tridimensionales**: son mapas de bits que guardan la información del color de cada píxel de la superficie del objeto y de su volumen. Esta técnica necesita un almacenamiento en disco bastante elevado, ya que tiene que almacenar la totalidad de los píxeles de un objeto.

Las texturas bidimensionales fueron las elegidas por ser idóneas para la implementación de un videojuego. Una vez escogido el tipo de textura a utilizar, se comenzó con la implementación de dichos métodos.

A continuación se detallan los pasos necesarios:

1. Cargar texturas en memoria

   Antes de cargar una imagen en memoria el usuario deberá verificar que las dimensiones de la misma sean potencia de dos, que el ancho sea igual al alto, es decir, sean cuadradas, y que el formato sea RGB. Si se desea cargar una imagen con color indexado, como por ejemplo una imagen en formato GIF, se deberá pasar a formato RGB antes de comenzar con el proceso de carga.

   El sistema se encarga de convertir las imágenes de formato ARGB a formato RGBA y guardar los valores de cada píxel de la imagen resultante, en un buffer de enteros. Para ello hace uso de un método llamado *extraer*, al que se le pasa como parámetro un mapa de bits y devuelve el buffer de enteros anteriormente mencionado.

2. Cargar las texturas en OpenGL

   Una vez cargada la textura en memoria RAM, se deberá pasar a la memoria usada por OpenGL ES 1.0 para el almacenamiento de imágenes, estos pasos se realizan en el método *load* [50]:

   - Obtener un mapa de bits, a partir de una imagen localizada en la carpeta res/drawable-hdpi, para transformarlo en una textura mediante la sentencia:

     ```
     Bitmap bmp = BitmapFactory.decodeResource (Resources res, int id);
     ```

     *Donde **res** es el recurso de la aplicación que contiene los datos de la imagen deseada e **id** el identificador de dichos datos.*

   - Asignar a cada textura cargada en memoria un identificador propio que se utilizará cuando llegue el momento de dibujarla sobre la superficie de un objeto.

     ```
     gl.glGenTextures (int n, int [] textures, int
     ```

     *Donde **n** indica el número de texturas que el usuario quiere generar y texturas el array de identificadores que corresponden a cada una de las texturas generadas.*

   - Establecer una textura destino entre una de las texturas contenidas en el array, para que OpenGL ES 1.0 trabaje con ella. De esta manera, cada vez que se desee modificar una propiedad sólo afectará a la textura con la que se esté trabajando en ese momento.

     ```
     gl.glBlindTexture (int target, int texture);
     ```

     *Donde **target** se corresponde con el tipo de textura (GL10. GL_TEXTURE_1D, GL10. GL_TEXTURE_2D, GL10. GL_TEXTURE_3D) y **texture** con la textura elegida.*

   - Pasar la textura a la memoria usada por OpenGL ES 1.0:

     ```
     gl.glTexImage2D (int target, int level, int internalformat, int
     width, int height, int border, int format, int type, Buffer
     pixels);
     ```

Normalmente cuando el tamaño de una imagen no se corresponde con el tamaño de un objeto, se usan filtros para evitar la pérdida de realismo de la escena, sin embargo hay una forma de adaptar el tamaño de la textura en función de las dimensiones y la lejanía del objeto, que mejora notablemente el rendimiento de la aplicación, los MipMaps, texturas de una misma imagen con tamaños inferiores al original. OpenGL se encarga de seleccionar el MipMap que mejor se adapte al objeto.

*En este método **target** se corresponde con el tipo de textura utilizada, **level**, con el nivel de MipMaps, es decir, el número de imágenes de menor tamaño que se generan a partir de la imagen original (que será 0 para la librería creada, ya que los objetos siempre se van a ver aproximadamente desde la misma distancia), **internalformat** con el modelo de color de la textura almacenada en la memoria RAM del ordenador (GL10.GL_RGBA, GL10.GL_RGB…), **width** con el ancho de la imagen, **height** con el alto y **border** con el borde, **format** con el formato con el que se almacena la textura en la memoria de OpenGL (GL10.GL_RGBA GL10.GL_RGB…), **type** con el tipo de variables usadas para almacenarla (GL10.GL_UNSIGNED_BYTE, GL10.GL_SHORT , GL10.GL_INT…) y **pixels** con el buffer de datos donde ha sido almacenada la información de dicha imagen.*

3. Parámetros de las texturas

OpenGL ofrece al usuario la posibilidad de modificar ciertas propiedades de cada una de las texturas. Dichas propiedades se configuran en el método *load* y son las siguientes:

- Filtros de visualización:

  Una textura es un mapa de bits formado por un conjunto de píxeles dispuestos de manera regular. Si una textura tiene un tamaño inferior o superior a la superficie en la cual se desea proyectar, se escalará, produciendo un aumento o una disminución del tamaño de cada píxel, que afectará de manera considerable al realismo de la escena. Para evitar este efecto, OpenGL permite añadir filtros de visualización a todas las texturas.

  En caso de que se produzca un aumento del tamaño de la imagen original, se utiliza la siguiente sentencia para el uso de filtros:

```
gl.glTexParameterx (int target, GL10.GL_TEXTURE_MAG_FILTER,float param);
```

*Donde **target** se corresponde con el tipo de textura utilizada (GL10. GL_TEXTURE_1D, GL10. GL_TEXTURE_2D, GL10. GL_TEXTURE_3D) y **param** con el filtrado de la misma. En caso de que el usuario no desee utilizar ningún filtro lo indicará mediante el valor GL_NEAREST, en caso contrario utilizará el valor GL_LINEAR que hará un filtrado lineal de la textura mediante una interpolación de cada píxel en la superficie sobre la que dibuja.*

En caso de que se produzca una disminución del tamaño de la imagen original se hará uso de la siguiente sentencia:

```
gl.glTexParameterx (int target, GL10.GL_TEXTURE_MIN_FILTER,float param);
```

*Donde **target** se corresponde con el tipo de textura utilizada y **param** con el filtrado de la misma. Este caso ofrece un mayor número de posibilidades de filtrado para el usuario, ya que a parte del filtrado lineal (GL_LINEAR) y del no filtrado (GL_NEAREST) también se permite el uso de MipMaps.*

En el motor gráfico se hace uso de dos filtros lineales.

- Repeticiones:

Al texturizar una escena el modelador aplica una textura pequeña que se repite a lo largo de una superficie. OpenGL permite al usuario dibujar la textura una sola vez sobre la superficie de objeto o dibujarla n veces.

Para dibujarla una sola vez se utilizarán las siguientes sentencias:

```
gl.glTexParameterx(int target,GL10.GL_TEXTURE_WRAP_S,GL10.GL_CLAMP_TO_EDGE);
gl.glTexParameterx(int target,GL10.GL_TEXTURE_WRAP_T,GL10.GL_CLAMP_TO_EDGE);
```

La primera de ellas indicará el uso de un filtro para las filas y la segunda, el uso de un filtro para las columnas.

Para dibujarla n veces se utilizarán las sentencias:

```
gl.glTexParameterx (int target, GL10.GL_TEXTURE_WRAP_S, GL10.GL_REPEAT);
gl.glTexParameterx (int target, GL10.GL_TEXTURE_WRAP_T, GL10.GL_REPEAT);
```

En el motor gráfico no se hace uso de esta técnica.

4. Proceso de blending y masking [15]

4.1 Blending

El proceso de blending es utilizado para crear objetos con efectos de transparencia haciendo uso del canal alpha. Este efecto se consigue combinando un primer plano translúcido, con un color de fondo para crear una mezcla intermedia.

Para llevar a cabo este proceso es necesario seguir los siguientes pasos:

- Activar el estado de blending mediante la sentencia:

```
gl.glEnable (GL10.GL_BLEND);
```

- Seleccionar el tipo de función a utilizar para combinar los componentes del píxel de origen con los del píxel de destino, almacenados en el framebuffer, mediante la sentencia:

```
gl.glBlendFunc (GL10 origen, GL10 destino);
```

*Donde **origen** indica cómo usar el factor blending o de transparencia de la fuente y el factor **destino** como usar el factor blending del destino.*

Los posibles tipos de funciones a utilizar son los siguientes:

Función	**Factor blending**
GL_ZERO	(0,0,0,0)
GL_ONE	(1,1,1,1)
GL_DST_COLOR	(Rd, Gd, Bd, Ad)
GL_SRC_COLOR	(Rs, Gs, Bs, As)
GL_ONE_MINUS_DST_COLOR	(1,1,1,1)-(Rd, Gd, Bd, Ad)
GL_ONE_MINUS_SRC_COLOR	(1,1,1,1)-(Rs, Gs, Bs, As)
GL_SRC_ALPHA	(As, As, As, As)
GL_ONE_MINUS_SRC_ALPHA	(1,1,1,1)-(As, As ,As, As)
GL_DST_ALPHA	(Ad, Ad, Ad, Ad)
GL_ONE_MINUS_DST_ALPHA	(1,1,1,1)-(Ad, Ad, Ad, Ad)
GL_SRC_ALPHA_SATURATE	(f,f,f,1); f=min(As,1-Ad)

Tabla 3: Funciones de blending

A continuación se muestran dos figuras; en la primera de ellas aparece un cubo con textura al que se le ha aplicado el proceso de blending, usando las funciones ***GL_ONE_MINUS_SRC_ALPHA*** *para el origen y* ***GL_ONE_MINUS_DST_ALPHA*** *para el destino y la segunda muestra un cubo con la función de blending deshabilitada.*

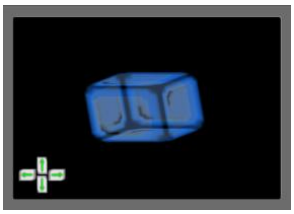


Figura 58: Cubo con efecto de blending

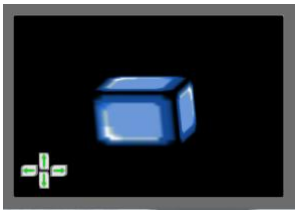


Figura 59: Cubo sin efecto de blending

4.2 Masking

Esta técnica consiste en dar un efecto de transparencia a determinadas partes de una imagen y es muy utilizada dentro del mundo de los videojuegos. En el motor gráfico se utiliza a la hora de dibujar el personaje principal, los enemigos y los objetos que forman el escenario. De este modo se consigue que alrededor de dichas imágenes no aparezca un cuadrado de color negro.

Para hacer uso de esta técnica se utilizarán dos imágenes diferentes; una de ellas llamada **imagen fuente**, que almacena la imagen que el usuario desea mostrar y recortar, y la otra llamada **imagen máscara**, con dos colores, el color negro para las partes de la imagen fuente que el usuario desea que se vean y el color blanco para las partes de la imagen que el usuario desea recortar.

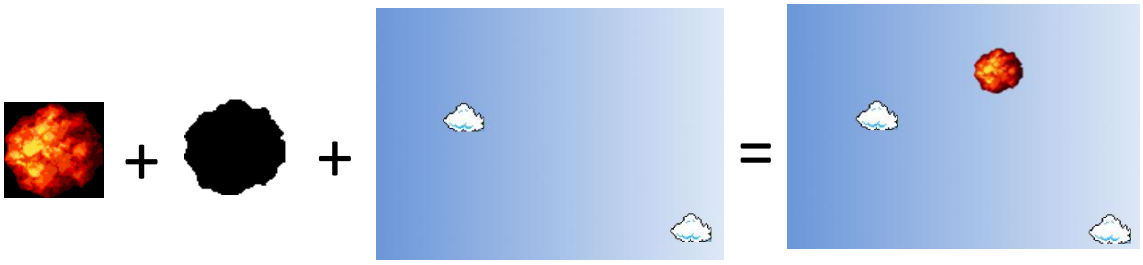


Imagen fuente | **Máscara** | **Fondo** | **Imagen recortada sobre fondo**

Figura 60: Masking

Para conseguir este efecto hay que seguir estos pasos:

1. Activar el efecto blending.
2. Seleccionar la función ***glBlendFunc (GL_DST_COLOR, GL_ZERO)***, que permite pintar en el destino los píxeles de color negro de la imagen original.
3. Pintar la figura con la máscara
4. Seleccionar la función ***glBlendFunc (GL_ONE, GL_ONE)***, que copia todos los píxeles de la imagen fuente ignorando los de color negro.
5. Pintar la figura con la imagen fuente.

```
gl.glEnable (GL10.GL_BLEND);
gl.glBlendFunc (GL10.GL_DST_COLOR, GL10.GL_ZERO);
fuente.dibuja (gl);
gl.glBlendFunc (GL10.GL_ONE, GL10.GL_ONE);
mascara.dibuja (gl);
```

Código 15: Efecto de masking

Una vez creada la clase ***Texturas***, se implementan las clases ***CuadradoTextura, CuboTextura***, ***PiramideTextura*** y ***TrianguloTextura,*** compuestas por los métodos *dameVertices, dameIndices* y *dibuja.* En el constructor de todas estás clases, se implementa un array con las posiciones de los téxtels del mapa de texturas, que se asignan a los vértices de las figuras.

Las dimensiones del espacio de textura están normalizadas, de manera que todas sus coordenadas varían entre 0 y 1. Para el cuadrado y el cubo las coordenadas asignadas se corresponden con las de la primera imagen de la figura 61, y para el triángulo y la pirámide, con las de la segunda imagen de esa misma figura.

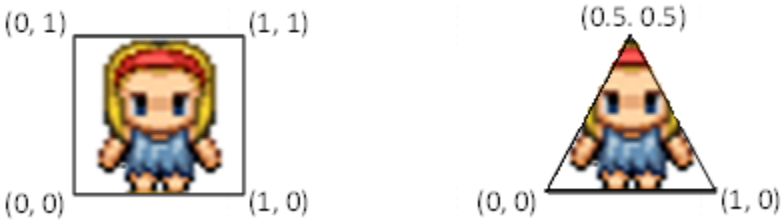


Figura 61: Coordenadas textura

Para configurar y crear cualquiera de estas figuras, se deberá crear un objeto de la clase ***FiguraTextura*** y pasarle 6 parámetros al constructor; el primero de ellos corresponderá con una instancia del objeto de la clase GL10, el segundo indicará al sistema la vista sobre la cual dibujar, el tercero el recurso donde se encuentra guardada la imagen y los tres últimos indicarán las dimensiones de la figura en números enteros, de tal manera que un 1 en números reales corresponderá con un 2^{16} en números enteros.

A continuación se muestra un ejemplo del código necesario:

```
private void inicializarObjetos (GL10 gl)
{
      piramide =new PiramideTextura (gl, view.getContext (),
      R.drawable.paisaje, 58000, 52000, 58000);
      cubo =new CuboTextura (gl, view.getContext (), R.drawable.paisaje,
      36384, 62768, 32000);
}
```

Código 16: Dibujar figuras con texturas

4.2.5.1.- Otras figuras con textura

Además de las figuras con texturas explicadas anteriormente, el usuario podrá hacer uso de otros dos tipos diferentes de figuras con texturas.

4.2.5.1.1.- Figuras con el efecto de blending y masking

Si el usuario desea aplicar una imagen sobre las superficies de las figuras que forman el videojuego de plataformas en 3D, y además desea que el fondo de la misma sea transparente, deberá hacer uso de la clase abstracta ***FiguraTransp***, y de las clases ***CuboTransp***, ***CuadradoTransp***, ***PiramideTransp*** y ***TrianguloTransp***.

La clase ***FiguraTransp*** hereda de la clase ***Figuras***, y en su constructor hace una llamada a la clase Texturas, explicada en el apartado 4.2.5, y almacena de manera temporal la información de los vértices y de los índices de las figuras, en dos buffers de datos, tal y como se explica en el apartado 4.2.2.

Las clases ***CuboTransp***, ***CuadradoTransp***, ***PiramideTransp*** y ***TrianguloTransp***, heredan de la clase ***FiguraTransp*** e implementan en su interior los métodos *dameVertices*, *dameIndices*, conforme a lo explicado en el apartado 4.2.2, y el método *dibuja,* que se encarga de dibujar las figuras haciendo uso de las técnicas de blending y masking, explicadas con anterioridad.

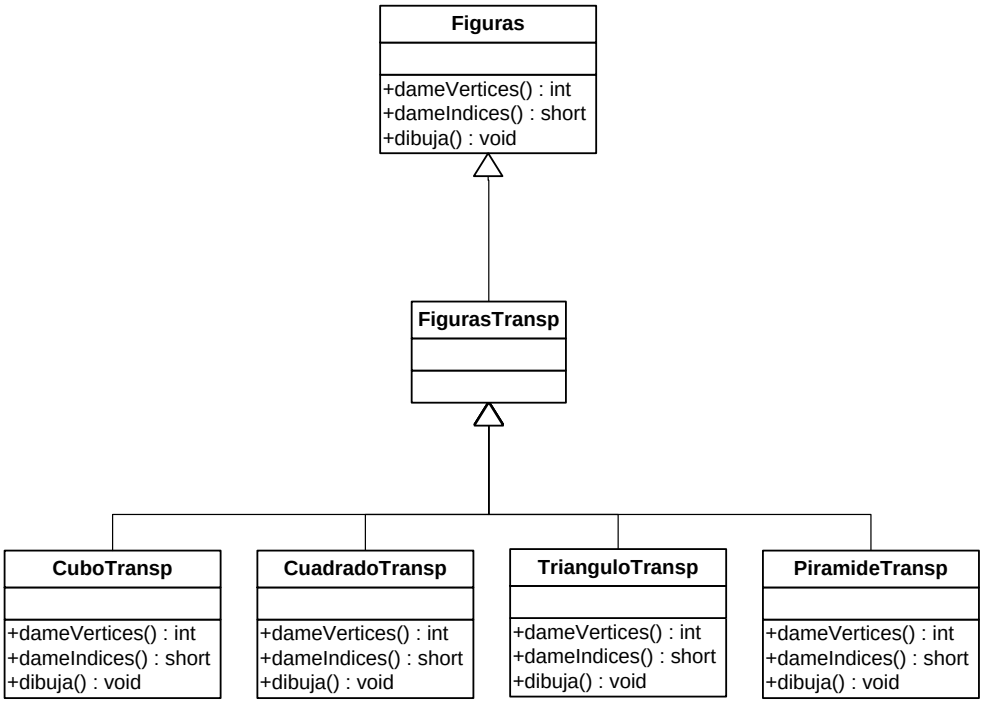


Figura 62: Diagrama de clases FiguraTransp

Para configurar y crear cualquiera de estas figuras, se deberá crear un objeto de la clase ***FiguraTransp*** y pasarle 7 parámetros al constructor; el primero de ellos corresponderá con una instancia del objeto de la clase GL10, el segundo indicará al sistema la vista sobre la cual dibujar, el tercero y el cuarto el recurso donde se encuentran guardadas la imagen fuente y la imagen máscara, y los tres últimos indicarán las dimensiones de la figura en números enteros, de tal manera que un 1 en números reales corresponderá con un 2^{16} en números enteros.

A continuación se muestra un ejemplo del código necesario:

```
private void inicializarObjetos (GL10 gl)
{
      piramide =new PiramideTextura (gl, view.getContext (),
      R.drawable.paisaje, 58000, 52000, 58000);
      cubo =new CuboTextura (gl, view.getContext (), R.drawable.fuente,
      R.drawable.mascara, 36384, 62768, 32000);
}
```

Código 17: Dibujar figuras con efecto de blending y masking

4.2.5.1.1.- Figuras con múltiples texturas

Si el usuario desea crear un cubo con una textura distinta en cada cara del mismo, deberá hacer uso de la clase ***CuboMTextura***, dicha clase hereda de la clase ***Figuras*** y está compuesta por un constructor, que se encarga de cargar la información de las 6 texturas necesarias para mapear el cubo llamando al método ***load*** de la clase ***Texturas*** y de rellenar el buffer de vértices y de texturas, y de un método denominado ***dibuja,*** que se encargará de dibujar cada cara con su textura correspondiente.

A dicho constructor hay que pasarle como parámetro una instancia del objeto de la clase GL10, la vista sobre la cual se desea dibujar dicha figura, seis recursos de las imágenes que se deseen cargar sobre las superficies de cada cara en el orden: cara delantera, cara izquierda, cara trasera, cara derecha, cara superior y cara inferior y el tamaño del ancho, alto y largo en números enteros.

```
private void inicializarObjetos (GL10 gl)
{
      cuboMTextura= new CuboMTextura (gl, view.getContext (),
      R.drawable.delantera, R.drawable.izq, R.drawable.trasera,
      R.drawable.derecha, R.drawable.superior, R.drawable.inferior,
      68000,68000, 68000);
}
```

Código 18: Dibujar cubo con múltiples texturas

Una vez explicadas las distintas figuras que se pueden utilizar para la construcción de la escena se procederá con una explicación de los tres elementos restantes, básicos y necesarios para la creación de un videojuego de plataformas; el escenario sobre el cual se desarrolla el videojuego, los personajes principales que maneja el usuario para alcanzar el objetivo final y los enemigos que añaden una dificultad extra al juego obstaculizando el camino hacia la meta.

4.2.6.- Dibujar el escenario.

Antes de comenzar a diseñar el escenario el usuario tendrá que valorar qué tipo de figuras desea utilizar para su construcción. Especificando las propiedades de la luz y de los materiales, los colores en cada vértice de las figuras o las imágenes deseadas para la texturización de cada elemento de la escena.

Una vez realizada esta tarea, el usuario deberá llevar a cabo dos pasos fundamentales:

1. **Construir una matriz tridimensional de números enteros denominada *mundo*** que represente el escenario, colocando en cada posición de la misma un número comprendido entre 0 y el número de elementos que forman la escena. De tal modo que si el usuario desea que en una posición del escenario el motor gráfico no dibuje nada, deberá rellenar dicha matriz con un 0 y si desea dibujar un cuadrado, un triángulo, un cubo, una pirámide o una esfera, deberá rellenar la misma con números comprendidos en un intervalo del 1 al número de elementos totales. Dicha matriz se deberá rellenar en el cuerpo de un método denominado *cargaMatriz*.

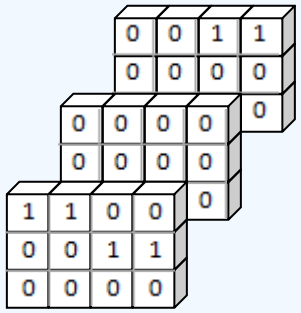
   A continuación se muestra un ejemplo del código utilizado para inicializar la matriz *mundo* y rellenarla:

```
public void inicializaMatriz ()

{
	for (int x = 0; x < mundo.length; x++)
	{
		for (int y = 0; y < mundo[x].length; y++)
		{
			for (int z = 0; z < mundo[x] [y].length; z++)
			{
				mundo [x] [y] [z] = 0;
```

Código 19: Construir matriz tridimensional mundo

```
public void cargaMatriz ()
{
	for (int base = 0; base < 2; base++)
	{
		mundo [base] [2] [0] = 1;
	}
	for (int x = 2; x < 4; x++)
	{
		mundo [x] [1] [0] = 1;
		mundo [x] [2] [2] = 2;
	}
}
```



Código 20: Cargar matriz tridimensional mundo

2. **Rellenar un vector con los objetos de las figuras que forman la escena, en el cuerpo de un método llamado *inicializaObjetos,* mediante el uso del método *asignarFigura* al que se le pasa cada figura creada como parámetro.**

   - En caso de que el usuario desee crear el escenario a partir de figuras compuestas por materiales, deberá crear las figuras tal y como se explica en el apartado 4.2.3. y a continuación añadir las mismas al vector.

   - En caso de que el usuario desee crear el escenario a partir de figuras a color, deberá crear las figuras tal y como se explica en el apartado 4.2.4. y a continuación añadir las mismas al vector.

   - En caso de que el usuario desee crear el escenario a partir de figuras con texturas, deberá guardar las imágenes en la carpeta *res/drawable-hdpi*, crear las figuras tal y como se explica en el apartado 4.2.5. y a continuación añadir las mismas al vector.

A continuación se muestra un ejemplo del código utilizado para rellenar un vector con dos tipos de figuras; un cubo con una textura y un cuadrado con una textura transparente.

```
public void inicializaObjetos (GLView view, GL10 gl)
{
      asignarFigura (new CuboTextura (gl, view.getContext (),
      R.drawable.imagen1, 38000, 32000, 38000));
      asignarFigura (new CuadradoTransp (gl, view.getContext (),
      R.drawable.fuente, R.drawable.mascara, 65536, 65536,0));
}
```

Código 21: Asignar figuras al escenario

El sistema utilizará la matriz creada por el usuario para dibuja el escenario. Dicha matriz se recorrerá de principio a fin siguiendo durante este recorrido varios pasos:

1. Trasladar el centro del sistema de coordenadas sobre el cual se encarga de pintar OpenGL hasta la posición actual de la matriz.

2. Dibujar en caso de que la distancia del objeto al punto de enfoque de la cámara sea menor o igual a un radio establecido de acuerdo con las siguientes condiciones:

    a. **Si es un 0:** no pintar nada.

    b. **Si es un número comprendido entre 1 y el número total de elementos en el vector:** pintar la figura situada en la posición del vector indicada por el número que contiene la matriz.

3. Trasladar el centro del sistema de coordenadas sobre el cual se encarga de pintar OpenGL hasta la posición inicial.

```
if (mundo[x] [y] [z]!= 0)
{
      if ((Math.abs(x - X) <= RADIO) && (Math.abs(y - Y) <=RADIO * 1.5f))
       {
            if((mundo[x][y][z]>0)&&((mundo[x][y][z])<= figuras.size()))
            {
                  figura = figuras.elementAt ((mundo[x][y][z])-1);
                  gl.glTranslatef(x, y, -z);
                  figura.dibuja (gl);
                  gl.glTranslatef (-x, -y, z);
            }
      }
}
```

Código 22: Dibujar escenario

4.2.6.1- Dibujar el fondo

Si el usuario desea dibujar una imagen de fondo en el videojuego deberá implementar el método abstracto inicializaFondo () de la clase Escenario. Este método recibe como parámetro dos variables; la primera de ellas indica al sistema la vista sobre la cual dibujar y la segunda corresponde con una instancia del objeto de la clase GL10 y devuelve un objeto de la clase Figuras.

A continuación se muestra un ejemplo del código necesario para dibujar un cuadrado con una textura de fondo:

```
public Figuras inicializaFondo(GLView view, GL10 gl)
{
      Figuras fondo= new CuadradoTextura (gl, view.getContext (),
      R.drawable.paisaje1, 1080000, 1690000, 0);
      return fondo;
}
```

Código 23: Dibujar fondo

4.2.7.- Personajes del videojuego

El sistema permite al usuario dibujar una serie de personajes con unas características propias. Dichas características, como la forma de desplazarse por el escenario o de interactuar unos con otros, serán configuradas por el usuario.

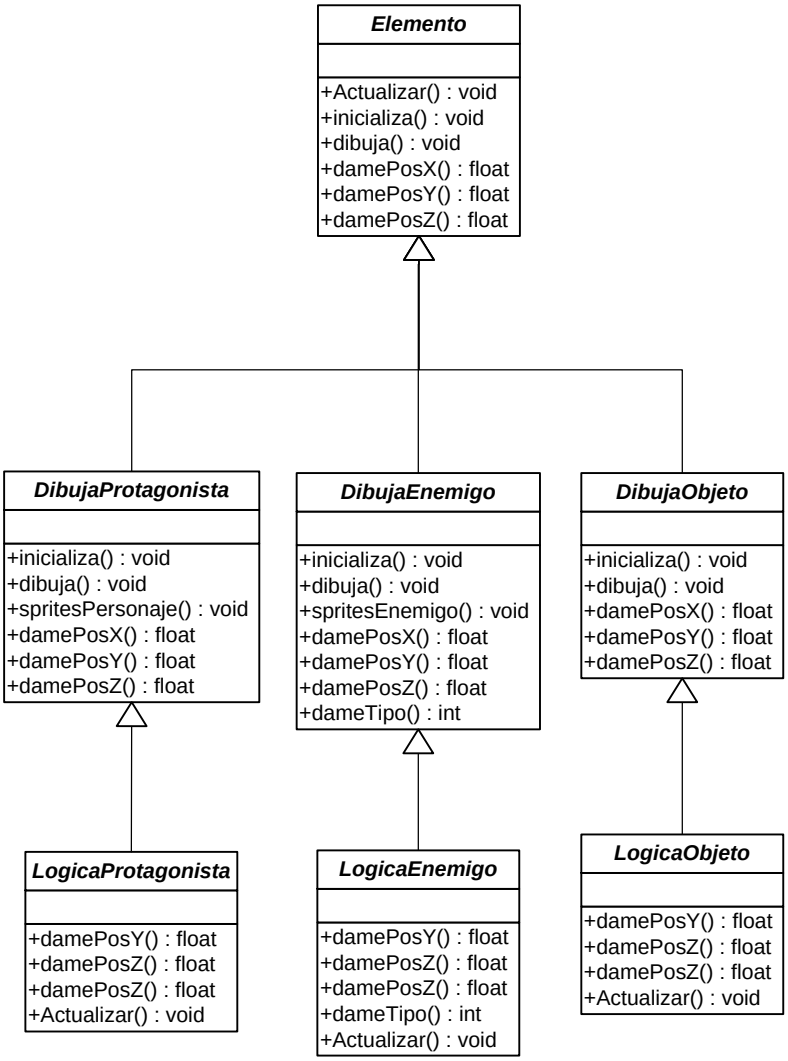


Figura 63: Diagrama de clases personajes del videojuego

4.2.7.1.- Personaje principal

El usuario podrá insertar en el videojuego un personaje principal, creando una clase que herede de ***DibujaProtagonista*** e implemente en su interior una serie de métodos predefinidos. Dichos métodos son los siguientes:

- **public float damePosX ():** devuelve la posición del personaje en el eje x.
- **public float damePosY ():** devuelve la posición del personaje en el eje y.
- **public float damePosZ ():** devuelve la posición del personaje en el eje z.
- **public void Actualiza ():** actualiza la posición del personaje.

Una vez declarados estos métodos, el usuario elegirá como desea dibujar al personaje principal del juego. Para representar al mismo mediante el uso de imágenes animadas, será necesario añadir en la carpeta *res/drawable-hdpi* una imagen llamada ***personaje***, con unas dimensiones de 192*256 bits, formada por 12 imágenes de 64*64bits correspondientes a cada movimiento del personaje.

Todas las filas de dicha composición, estarán formadas por grupos de tres imágenes que representan el movimiento del personaje desplazándose hacia adelante, hacia la derecha, hacia la izquierda y hacia atrás respectivamente.

Además si el usuario desea recortarla para hacer el fondo transparente, deberá añadir a la carpeta *res/drawable-hdpi* la máscara correspondiente a la imagen fuente y llamarla ***mascarap***. Dicha máscara tendrá unas dimensiones de 192*256 bits y estará formada por 12 imágenes de 64*64bits correspondientes a cada estado del personaje.

A continuación se muestra un ejemplo de una imagen fuente utilizada para la caracterización de un personaje y la máscara correspondiente a la misma.

Figura 64: Imagen personaje

Figura 65: Imagen mascarap

Una vez realizados estos pasos, el sistema obtendrá la información de dichas imágenes, dibujará el personaje principal y lo actualizará en cada paso, mediante las clases ***SpritesProtagonista, DibujaPersonaje y ObjPersonaje***.

El objetivo de la clase ***SpritesProtagonista*** es cargar en la memoria usada por OpenGL ES 1.0 para el almacenamiento de imágenes, las imágenes fuente y máscara.

Esta clase está formada por 4 métodos:

- ***extract:*** obtiene el mapa de bits de la imagen, que se le pasa como parámetro a dicho método.

- ***cortarMascara:*** almacena en un array de 4 filas y 3 columnas llamado ***mascaracortada***, 12 texturas que contienen información acerca de las máscaras correspondientes a cada estado del personaje.

  Para ello se obtiene el mapa de bits de la imagen guardada por el usuario en la carpeta *res/drawable-hdpi*, se calculan el ancho y el alto de las imágenes individuales y se crean una por una, las 12 texturas que representan los sprites de la máscara del personaje.

```
public static void cortarMascara (int resource, Context context)
{
	Bitmap bmp = BitmapFactory.decodeResource (context.getResources
	(), resource);
	ancho =bmp.getWidth () / BMP_COLUMNS;
	alto = bmp.getHeight () / BMP_ROWS;

	for (int filas=0; filas<BMP_ROWS; filas++)
	{
		for (int columnas=0; columnas<BMP_COLUMNS; columnas++)
		{
			mascaracortada [filas] [columnas] =
			Bitmap.createBitmap (bmp, (x + ancho * columnas),
			(y+alto* filas), ancho, alto);
		}
	  }
}
```

Código 24: Cortar máscara personaje

- ***cortarImagen:*** se encarga de almacenar en un array de 4 filas y 3 columnas llamado ***imagencortada***, 12 texturas que representan los sprites del personaje.

  Para ello se obtiene el mapa de bits de la imagen guardada por el usuario en la carpeta *res/drawable-hdpi*, se calculan el ancho y el alto de las imágenes individuales y se crean una por una, las 12 texturas.

- ***cargarImagen*** se encarga de asignar a cada textura cargada en memoria un identificador propio, de establecer una textura destino entre una de las texturas contenidas en el array, de pasar la textura a la memoria usada por OpenGL y de configurar los filtros de visualización.

  Para ello es necesario pasarle a dicho método 6 parámetros; una instancia del objeto de la clase GL10, una vista, un recurso, una fila y una columna que indiquen el sprite del personaje a dibujar, y un número que en caso de ser uno indica que la imagen a cargar es de tipo fuente y en caso contrario, que es de tipo máscara.

A continuación se muestra parte del código utilizado en este método.

```
public static int cargarImagen (GL10 gl, int resource, Context
context, int fila, int columna, int i)
{
      if (i==1)
      {
            ByteBuffer bb = extract (imagencortada [fila] [columna]);
            int [] tmp_tex = new int [1];
            gl.glGenTextures (1, tmp_tex, 0);
            int tex = tmp_tex [0];
            gl.glBindTexture (GL10.GL_TEXTURE_2D, tex);
            gl.glTexImage2D (GL10.GL_TEXTURE_2D, 0, GL10.GL_RGBA,
            imagencortada [fila] [columna] .getWidth (),
            imagencortada [fila] [columna].getHeight (), 0,
            GL10.GL_RGBA, GL10.GL_UNSIGNED_BYTE, bb);
            gl.glTexParameterx (GL10.GL_TEXTURE_2D,
            GL10.GL_TEXTURE_MIN_FILTER, GL10.GL_LINEAR);
            gl.glTexParameterx (GL10.GL_TEXTURE_2D,
            GL10.GL_TEXTURE_MAG_FILTER, GL10.GL_LINEAR);
            return tex;
      }
```

Código 25: Cargar imagen personaje

La clase ***DibujaPersonaje*** se encarga de crear los objetos que representan al personaje, de actualizar los sprites [68] de las superficies de dichos objetos y de dibujarlos.

La clase está formada por 4 métodos:

- ***inicializa:*** este método se encarga de cortar las imágenes ***personaje*** y ***mascarap*** y de inicializar los objetos necesarios para representar al personaje principal. Dichos objetos sumarán un total de 24.
    - Los 12 primeros se almacenarán en un array de 4 filas y 3 columnas llamado ***mascara***, y corresponderán con 12 cuadrados sobre los que se mapearán las texturas resultantes de cortar la máscara del personaje.
    - Los 12 últimos se almacenarán en un array de 4 filas y 3 columnas llamado ***personaje****, y corresponderán* con 12 cuadrados sobre los que se mapearán las texturas resultantes de cortar la imagen fuente que representa a los sprites del personaje principal.

```
public void inicializa (GLView view, GL10 gl)
{
      SpritesProtagonista.cortarImagen (R.drawable.personaje,
      view.getContext ());
      SpritesProtagonista.cortarMascara (R.drawable.mascarap,
      view.getContext ());

      for (int filas = 0; filas < 4; filas++)
      {
            for (int columnas = 0; columnas < 3; columnas++)
            {
                  mascara [filas] [columnas]= new ObjPersonaje
                  (gl, view.getContext (), R.drawable.mascarap,
                  filas, columnas, 2);
                  personaje [filas] [columnas] = new
                  ObjPersonaje (gl, view.getContext (),
                  R.drawable.personaje, filas, columnas, 1);
            }
      }
}
```

Código 26: Inicializar personaje

- ***spritesPersonaje:*** este método se encarga de actualizar los sprites del personaje y de simular un movimiento real y animado del mismo. Para ello se hace uso de 7 variables; la primera de ellas llamada ***movimentoPersonaje*** que se encargan de comunicar al sistema el cambio de movimiento del personaje principal del juego mediante la actualización de la variable entera ***columna***, y las restantes llamadas ***spriteAb***, ***spriteAr, spriteIzq y spriteDer*** que se encargan de comunicar al sistema el cambio de dirección del personaje principal del juego mediante la actualización de la variable entera ***fila***.

  Para conseguir este efecto es necesario que el usuario actualice dichas variables: en primer lugar deberá actualizar las variables booleanas ***spriteAb***, ***spriteAr, spriteIzq y spriteDer*** cada vez que el personaje cambie de dirección. Una vez realizada esta tarea, deberá incrementar el valor de la variable entera ***movimentoPersonaje*** en 1 cada vez que el personaje principal avance, independientemente de la dirección del mismo.

  Por ejemplo, si el personaje avanza en el eje positivo de las x, es decir, hacia la derecha, el usuario tendrá que dar el valor *false* a las variables ***spriteAb***, ***spriteAr, spriteIzq*** y el valor *true* a la variable ***spriteDer.*** Y a continuación incrementar en 1 el valor de ***movimentoPersonaje.***

- ***dibuja:*** este método se encarga de dibujar al personaje principal en la posición del escenario adecuada, mediante el uso de los métodos ***damePosX ()***, ***damePosY ()*** y d***amePosZ ()***. También se encarga de actualizar los sprites que representan al mismo mediante las variables ***fila*** y ***columna***, anteriormente mencionadas. Además añade realismo a la escena recortando las imágenes mediante las técnicas de blending y masking y habilitando el efecto de profundidad.

```
public void dibuja (GL10 gl)
{
	gl.glEnable (GL10.GL_DEPTH_TEST);
	gl.glEnable (GL10.GL_BLEND);
	gl.glTranslatef (damePosX (), damePosY () +1,-30.0f-damePosZ());
	gl.glBlendFunc (GL10.GL_DST_COLOR, GL10.GL_ZERO);
	mascara [fila] [columna].draw (gl);
	gl.glBlendFunc (GL10.GL_ONE, GL10.GL_ONE);
	personaje [fila] [columna].draw (gl);
}
```

Código 27: Dibujar personaje

La clase ***ObjPersonaje*** se encarga de crear un cuadrado con textura, sobre el cual se mapeará cada sprite del personaje o de la máscara.

A continuación se muestra el resultado de las animaciones correspondientes al desplazamiento del personaje en las distintas posiciones de la pantalla, haciendo uso de las imágenes 61 y 62:

Desplazamiento frontal: el personaje se mueve en sentido positivo al eje z.

Figura 66: Tira de sprites personaje desplazándose hacia el frente

Desplazamiento lateral: el personaje avanza en el eje positivo de las x.

Figura 67: Tira de sprites personaje desplazándose hacia la derecha

Desplazamiento lateral: el personaje avanza en el eje negativo de las x.

Figura 68: Tira de sprites personaje desplazándose hacia la izquierda

Desplazamiento hacia el fondo: el personaje se mueve en sentido negativo al eje z.

Figura 69: Tira de sprites personaje desplazándose hacia el fondo

4.2.7.2.- Dibujar enemigos del videojuego.

El usuario podrá insertar en el videojuego, uno o varios enemigos, mediante la implementación de una clase que herede de la clase ***DibujaEnemigo*** e implemente en su interior una serie de métodos predefinidos. Dichos métodos son los siguientes:

- **public float damePosX ():** devuelve la posición del enemigo en el eje x.
- **public float damePosY ():** devuelve la posición del enemigo en el eje y.
- **public float damePosZ ():** devuelve la posición del enemigo en el eje z.
- **public int dameTipo ():** devuelve un entero correspondiente a cada tipo de enemigo. El usuario podrá diseñar hasta tres tipos de enemigos distintos, de tal manera que cada uno tenga un comportamiento distinto.
- **public void Actualizar ():** actualiza la posición de los enemigos.

Una vez declarados estos métodos, el usuario elegirá como desea dibujar los distintos enemigos del juego. Para representarlos mediante el uso de imágenes animadas, será necesario añadir en la carpeta *res/drawable-hdpi* una imagen llamada ***malo1*** para el tipo 1, ***malo2*** para el tipo 2 y ***malo3*** para el tipo3, todas ellas con unas dimensiones de 192*256 bits, formadas por 12 imágenes de 64*64bits correspondientes a cada estado del enemigo.

Todas las filas de dicha composición, estarán formadas por grupos de tres imágenes que representan el movimiento del enemigo desplazándose hacia adelante, hacia la derecha, hacia la izquierda y hacia atrás respectivamente.

Además si el usuario desea recortarlas para hacer el fondo transparente, deberá añadir a la carpeta *res/drawable-hdpi* las máscaras correspondientes a cada enemigo, llamando ***mascaram1***, para la imagen fuente ***malo1***, ***mascaram2***, para la imagen fuente ***malo2*** y ***mascaram3***, para la imagen fuente ***malo3***. Dichas máscaras tendrán unas dimensiones de 192*256 bits y estarán formadas por 12 imágenes de 64*64bits correspondientes a cada estado del personaje.

A continuación se muestra un ejemplo de una imagen fuente utilizada para la caracterización de un enemigo y la máscara correspondiente a la misma.

Figura 70: Imagen malo1

Figura 71: Imagen mascaram1

Una vez realizados estos pasos el sistema obtendrá la información de dichas imágenes, dibujará los enemigos y los actualizará en cada paso, mediante las clases ***SpritesEnemigo, DibujaEnemigo y ObjEnemigo***.

El objetivo de la clase ***SpritesEnemigo*** es cargar en la memoria usada por OpenGL para el almacenamiento de imágenes, las imágenes fuente y máscara.

Esta clase está formada por 4 métodos:

- ***extract:*** obtiene el mapa de bits de la imagen, que se le pasa como parámetro a dicho método.

- ***cortarMascara:*** almacena en un array tridimensional de dimensiones 4filas x 3 columnas x N número de enemigos llamado ***mascaraCortada***, todas las texturas que contienen información acerca de las máscaras de los distintos estados de los enemigos que componen el juego.

```
public static void cortarMascara (int resource, Context context)
{
      Bitmap bmp = BitmapFactory.decodeResource (context.getResources
      (), resource);

      ancho =bmp.getWidth () / BMP_COLUMNS;
      alto = bmp.getHeight() / BMP_ROWS;

      for (int i=0; i<DibujaEnemigo.numEnemigos; i++)
      {
            for (int filas=0; filas<BMP_ROWS; filas++)
            {
                  for (int columnas=0;columnas<BMP_COLUMNS;columnas++)
                  {
                        mascaraCortada [filas] [columnas] [i]=
                        Bitmap.createBitmap (bmp, (x + ancho *
                        columnas), (y+alto*filas), ancho, alto);
                  }
            }
       }
}
```

Código 28: Cortar máscaras enemigos

- ***cortarImagen:*** almacena en un array tridimensional de dimensiones 4filas x 3 columnas x N número de enemigos llamado ***mascaraCortada***, todas las texturas que contienen información acerca de las imágenes fuente de los distintos estados de los enemigos que componen el juego.

- ***cargarImagen*** se encarga de asignar a cada textura cargada en memoria un identificador propio, de establecer una textura destino entre una de las texturas contenidas en el array, de pasar la textura a la memoria usada por OpenGL y de configurar los filtros de visualización.

  Para ello es necesario pasarle a dicho método 7 parámetros; una instancia del objeto de la clase GL10, una vista, un recurso, una fila y una columna que indiquen el sprite del personaje a dibujar, un número que en caso de ser uno indica que la imagen a cargar es de tipo fuente y en caso contrario, que es de tipo máscara y otro número que indica el tipo de enemigo sobre el cual se va a trabajar.

La clase ***DibujaEnemigo*** se encarga de crear los objetos que representan al personaje, de actualizar los sprites de las superficies de dichos objetos y de dibujarlos.

La clase está formada por 4 métodos:

- ***inicializa:*** este método se encarga de cortar las imágenes y de inicializar los objetos necesarios para representar a los enemigos.

  - Para almacenar objetos cuadrados sobre los que se mapearán las texturas resultantes de recortar las imágenes máscara de los distintos tipos de enemigos, se creará un array llamado ***mascaras***, cuyas dimensiones serán N número de enemigos x 4 filas x 3 columnas.

  - Para almacenar objetos cuadrados sobre los que se mapearán las texturas resultantes de recortar las imágenes fuente de los distintos tipos de enemigos, se creará un array llamado ***enemigos***, cuyas dimensiones serán N número de enemigos x 4 filas x 3 columnas.

Una vez creados dichos array se procederá a rellenar los mismos. Para ello se recorrerán uno a uno los distintos enemigos que forman el juego, y se comprobará para cada uno de ellos el tipo de enemigo al que pertenecen. Si se trata de un enemigo de tipo 0, se recortarán las imágenes ***malo1*** y ***mascaram1*** y se añadirán a cada posición de los arrays, **mascaras** y **enemigos**, los 24 objetos cuadrados con cada una de las texturas resultantes. Si se trata de un enemigo de tipo 1 o 2 sucederá lo mismo pero las imágenes recortadas serán ***malo2*** y ***mascaram2*** o ***malo3*** y ***mascaram3***, respectivamente.

```
public void inicializa (GLView view, GL10 gl)
{
      enemigos     = new ObjEnemigo [numEnemigos] [4] [3];
      mascaras     = new ObjEnemigo [numEnemigos] [4] [3];
      columna      = new int [numEnemigos];
      fila         = new int [numEnemigos];
      movimientoEnemigo = new int [numEnemigos];

      for (int i = 0; i < numEnemigos ; i++)
      {
            if (dameTipo() == 0)
            {

                  SpritesEnemigo.cortarEnemigo (R.drawable.malo1,
                  view.getContext ());
                  SpritesEnemigo.cortarMascara
                  (R.drawable.mascaram1,  view.getContext());

                  for(int filas = 0; filas < 4; filas++)
                  {
                        for(int columnas=0; columnas< 3; columnas++)
                        {
                              mascaras [i][filas][columnas] = new
                              ObjEnemigo(gl,view.getContext(),R.draw
                              able.malo1,filas,columnas,i,1);
                              enemigos [i][filas][columnas] = new
                              ObjEnemigo(gl,view.getContext(),R.draw
                              able.mascaram1,filas,columnas,i,2);
                        }
                  }
            }
```

Código 29: Cargar imágenes enemigos

- ***spritesEnemigo:*** este método se encarga de actualizar los sprites de todos los enemigos y de simular un movimiento real y animado de los mismos.

  Para actualizar el cambio de movimiento de cada uno de los enemigos, será necesario el uso de dos arrays de enteros con tantas posiciones como número de enemigos haya en el juego. El primero de ellos llamado, ***movimentoPersonaje*** se encarga de comunicar al sistema el cambio de movimiento de cada uno de los enemigos y el segundo de ellos llamado ***columna***, se encarga de actualizar la columna de la textura a utilizar para cada enemigo.

  Para actualizar la dirección de cada uno de los enemigos, será necesario el uso de 4 variables llamadas ***spriteAb***, ***spriteAr, spriteIzq y spriteDer*** que se encargan de comunicar al sistema el cambio de dirección de cada uno de los enemigos y un array de enteros con tantas posiciones como número de enemigos haya en el juego llamado ***fila,*** que se encarga de actualizar la fila de la textura a utilizar para cada enemigo.

  Para conseguir este efecto es necesario que el usuario actualice las variables booleanas ***spriteAb***, ***spriteAr, spriteIzq y spriteDer*** cada vez que el enemigo cambie de dirección. Por ejemplo, si el enemigo 1 avanza en el eje positivo de las x, es decir, hacia la derecha, el usuario tendrá que dar el valor *false* a las variables ***spriteAb***, ***spriteAr, spriteIzq*** y el valor *true* a la variable ***spriteDer.***

- ***dibuja:*** este método se encarga de dibujar a cada enemigo en la posición del escenario adecuada, mediante el uso de los métodos ***damePosX ()***, ***damePosY ()*** y ***damePosZ ()***. También se encarga de actualizar los sprites que representan al mismo mediante las variables ***fila*** y ***columna***, anteriormente mencionadas. Además añade realismo a la escena recortando las imágenes mediante las técnicas de blending y masking y habilitando el efecto de profundidad.

La clase ***ObjEnemigo*** se encarga de crear un cuadrado con textura, sobre el cual se mapeará cada sprite del enemigo o de la máscara.

A continuación se muestra el resultado de las animaciones correspondientes al desplazamiento de un enemigo en las distintas posiciones de la pantalla, haciendo uso de las imágenes 67 y 68:

Desplazamiento frontal: el enemigo se mueve en sentido positivo al eje z.

Figura 72: Tira de sprites enemigo desplazándose hacia el frente

Desplazamiento lateral: el enemigo avanza en el eje positivo de las x.

Figura 73: Tira de sprites enemigo desplazándose hacia la derecha

Desplazamiento lateral: el enemigo avanza en el eje negativo de las x.

Figura 74: Tira de sprites enemigo desplazándose hacia la izquierda

Desplazamiento hacia el fondo: el enemigo se mueve en sentido negativo al eje z.

Figura 75: Tira de sprites enemigo desplazándose hacia el fondo

4.2.7.3.- Dibujar objetos del videojuego.

El usuario podrá insertar en el videojuego, uno o varios objetos que se muevan de un lado a otro de la pantalla, mediante la implementación de una clase que herede de la clase ***DibujaObjeto*** e implemente en su interior una serie de métodos predefinidos. Dichos métodos son los siguientes:

- **public float damePosX ():** devuelve la posición del enemigo en el eje x.
- **public float damePosY ():** devuelve la posición del enemigo en el eje y.
- **public float damePosZ ():** devuelve la posición del enemigo en el eje z.
- **public void Actualizar ():** actualiza la posición de los enemigos.

Una vez declarados estos métodos, el usuario elegirá como desea dibujar los distintos enemigos del juego. Para ello será necesario añadir en la carpeta *res/drawable-hdpi* una imagen fuente llamada ***objeto*** de dimensiones 64*64bits.

Además si el usuario desea recortarla para hacer el fondo transparente, deberá añadir a la carpeta *res/drawable-hdpi* la máscara correspondientes a dicha imagen fuente y llamarla ***objetomascara***.

CAPÍTULO 5.- PRUEBAS

5.1.- Pruebas gráficas

Antes de comenzar con las pruebas de jugabilidad se realizaron ciertas pruebas en los gráficos, el objetivo de las mismas era que el sistema dibujara las figuras indicadas por el usuario en la posición adecuada para que así, el jugador pudiera observar el videojuego en la pantalla del dispositivo móvil sin ningún problema.

5.1.1.- Fondo

Dibujar el fondo fue una tarea bastante complicada, ya que OpenGL ES 1.0 no permitía dibujar una imagen estática de fondo haciendo uso de la clase SurfaceView. Este inconveniente se solucionó permitiendo al usuario crear un rectángulo, o cualquier otra figura, de unas dimensiones proporcionales al tamaño de la pantalla del dispositivo móvil. Una vez creado, el sistema se encargaría de mapear sobre la superficie de la figura, una textura cualquiera y de desplazarla mediante movimientos de la cámara, a una posición lo suficientemente alejada del eje de coordenadas z para que así, no se solapara con ninguna otra figura del escenario ni con los personajes del juego.

5.1.2.- Cámara

Uno de los objetivos de este proyecto era implementar un motor gráfico que permitiera crear un videojuego de plataformas en 3D cuyo objetivo principal, fuera encontrar el camino apropiado para llegar al final del nivel mediante movimientos de la cámara.

Para hacer las pruebas de dibujado se tenía que comprobar si al realizar este movimiento de cámara la imagen de fondo, el escenario y los personajes principales no se veían afectados. Para comprobar esto en primer lugar se añadió un pad formado por 4 botones en la vista del videojuego.

Cuando el usuario de la aplicación pulsara la tecla superior del pad, la cámara se movería en el eje positivo de las y, cuando pulsara la tecla inferior, se movería en el eje negativo de las y, cuando pulsara la tecla derecha, se movería en el eje positivo de las x y cuando pulsara la tecla izquierda, se movería en el eje negativo de las x.

Mediante esta prueba se descubrió que al producirse un movimiento de cámara exagerado el fondo se superponía con algunos de los elementos del escenario y se observaba que la imagen de fondo no era lo suficientemente grande como para ocupar la totalidad de la superficie de la pantalla de juego. A continuación se muestra un gráfico que muestra estos efectos:

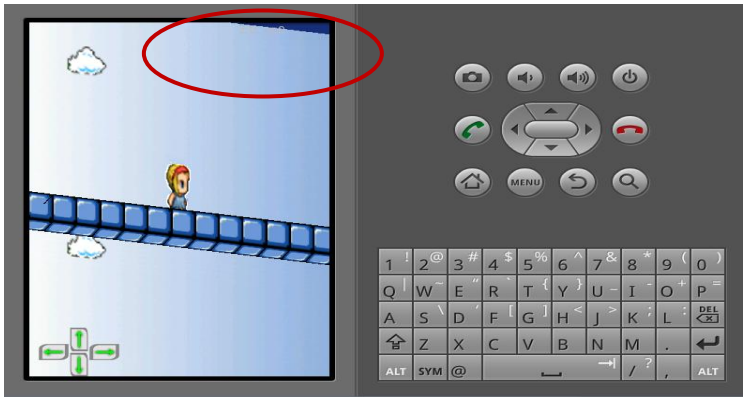


Figura 76: Prueba gráfica 1

Para evitar el problema del solapamiento se decidió utilizar el método **gl.glEnable (GL10.GL_DEPTH_TEST)**, de este modo el motor gráfico sólo dibujaría los puntos de la matriz de píxeles con menor profundidad y para evitar el problema del tamaño de la imagen de fondo, se decidió escalarla. A continuación se muestran dos gráficos del resultado final de las pruebas una vez realizados estos cambios:

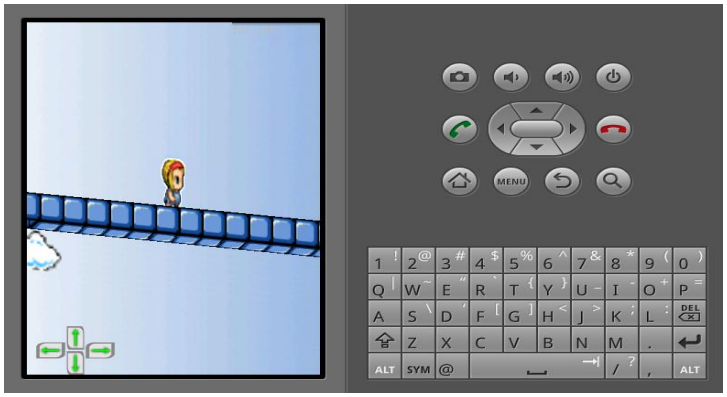


Figura 77: Prueba gráfica 2

5.2.- Pruebas de jugabilidad

Para que el usuario pueda diseñar un videojuego de plataformas en 3D que funcione correctamente, hay que comprobar que el número de frames[13] por segundo [14] tenga un valor aproximado de 25.

Por ello en el hilo[19] de ejecución del proyecto se va a calcular el número de fps que tarda el proyecto en realizar todas las tareas en su totalidad.

5.2.1.- Analizar el número de fps que el sistema tarda en dibujar el escenario

Una vez diseñadas todas las figuras, se procedió a crear una matriz, *mundo,* de prueba, bastante sencilla. A continuación se muestra el código utilizado para su implementación:

```
public void cargaMatriz ()
{
	for (int x = 0; x < 4; x++)
	{
		mundo [x][2][0] = 1;
	}
	for (int x = 4; x < 9; x++)
	{
		mundo [x][1][0] = 1;
		mundo [x][2][0] = 2;
	}
	for (int x = 9; x < 16; x++)
	{
		mundo [x][2][0] = 1;
	}
	for (int x = 4; x < 9; x++)
	{
		mundo [x][5][0] = 1;
	}
}
```

Código 30: Prueba matriz mundo

En primer lugar se decidió que la forma más adecuada para dibujar el escenario era recorrer dicha matriz y dependiendo del número que hubiera en su interior, crear una figura u otra. Este método no era el apropiado por dos razones:

- Al crear las figuras en el interior del motor gráfico, no se le daba la posibilidad al usuario de elegir las propiedades ni el tipo de las mismas.

  Por ese motivo, se decidió que el usuario creara un vector figuras. De este modo podría elegir las figuras que iba a utilizar para diseñar el escenario y las propiedades de las mismas (como la textura o el uso de máscaras).

- Al crear una nueva figura para cada posición de la matriz *mundo*, el número de fps calculado era inferior al deseado, aproximadamente 12, esto provocaba la ralentización del movimiento del videojuego.

  Para solventar este problema se decidió crear una figura para cada elemento del vector *figuras*, situada en la posición (0, 0, 0), recorrer la matriz *mundo* y dependiendo del número contenido en su interior, desplazar la figura correspondiente a dicho número hasta la posición indicada en la matriz, una vez desplazada, volver a desplazarla hasta su posición de origen. De este modo, sólo se crea una figura para cada elemento del vector y se amplia el número de fps de la aplicación.

5.2.2.- Analizar el número de fps que el sistema tarda en dibujar el personaje principal

Para probar el rendimiento del motor gráfico también fue necesario dibujar un personaje con sprites, dicho personaje tendría un movimiento constante en el eje positivo de las x. De este modo a parte del rendimiento, se comprobaría el funcionamiento de la actualización de los sprites del personaje. A continuación se muestra el código utilizado en el cuerpo de la clase ***LogicaProtagonista***:

```
public class LogicaProtagonista extends DibujaProtagonista
{

	private float	posX, posY, posZ;
	float		dX, dY, dZ;

	public LogicaProtagonista (float posX, float posY, float posZ)
	{
		this.dX = posX;
		this.dY = posY;
		this.dZ = posZ;

	}
```

```
	public void Mover (GL10 gl, GLView view)
	{
		spriteDer= false;
		spriteIzq= true;
		movimientoPersonaje++;
		dX = dX + 0.2f;
	}

	public void Actualizar ()
	{
		this.posX = dX;
		this.posY = dY;
		this.posZ = dZ;
	}

	public float damePosX ()
	{
		return posX;
	}

	public float damePosY ()
	{
		return posY;
	}

	public float damePosZ ()
	{
		return posZ;
	}
}
```

Código 31: Prueba lógica protagonista

Para actualizar el movimiento del mismo se decidió recortar de la textura compuesta por todos los estados del personaje, el sprite correspondiente con el sentido del mismo en cada paso y una vez recortada, crear un cuadrado con dicha textura. Este método era bastante ineficaz ya que ocasionaba dos grandes inconvenientes:

- El número de fps calculado era de 2 fps, un valor demasiado bajo que provocaba una ralentización del videojuego demasiado elevada.

  Este problema se solucionó inicializando una figura para cada estado del personaje y dibujando las mismas en el momento apropiado, es decir, dibujando la figura correspondiente al estado del personaje necesitado.

- Las texturas almacenadas en memoria aumentan cada vez que el sistema cambia el sprite del personaje, lo que producía un rendimiento bastante inadecuado del videojuego.

  Este problema se solucionó recortando todas las texturas de los sprites del personaje y de su máscara antes de crear las figuras para cada estado del mismo.

5.2.3.- Analizar el número de fps que el sistema tarda en dibujar los enemigos

Además de dibujar el personaje principal con sprites, también fue necesario dibujar un número elevado de enemigos con sprites. Dichos enemigos tendrían un movimiento bastante sencillo, seguirían al personaje principal en el eje positivo de las x. De este modo se comprobaría el rendimiento del videojuego con un número elevado de enemigos y el funcionamiento de la actualización de los sprites.

Para actualizar el movimiento de los enemigos se decidió recortar de las texturas compuestas por todos los estados de los enemigos, los sprites correspondientes con la dirección de los mismos en cada paso y una vez recortadas, crear tantos cuadrados como enemigos haya con la textura obtenida. Este método era ocasionaba dos grandes inconvenientes:

- El número de fps calculado era de 0.5 fps, un valor excesivamente bajo.

  Este problema se solucionó inicializando para cada enemigo una figura para cada sprite y dibujando las mismas en el momento apropiado. En vez de crear tantas figuras como cambios de estado experimentaran los enemigos, se crearía una figura para cada estado de los mismos.

- Las texturas almacenadas en memoria aumentaban cada vez que se actualizaban los sprites de los enemigos, lo que producía un rendimiento bastante inadecuado del videojuego.

  Este problema se solucionó recortando todas las texturas de los sprites de todos los enemigos y de sus máscaras antes de crear las figuras para cada estado.

Una vez realizadas todas estas pruebas se llegó a la conclusión de que tampoco era necesario dibujar la totalidad del escenario y de los personajes en él situados. Por ello se decidió dibujar exclusivamente los elementos y personajes situados a una distancia de 8,5f del punto de enfoque de la cámara, de este modo se conseguiría un aumento del número de fps totales.

Al terminar con todas las pruebas se comprobó el número de frames por segundo resultantes, dicho valor giraba en torno a los 64fps, un valor demasiado elevado que imposibilitaba la jugabilidad del videojuego. Por este motivo se decidió acotar los fps. En caso de que el valor fuera mayor a 30, se dormiría el hilo tantos fps como sobraran hasta llegar a ese número. A continuación se muestra el código utilizado para dicho cálculo:

```
long currentTime = System.currentTimeMillis();

if(currentTime-lastTime > 33)
{
	GLView.fps = frames * 1000.0 / (currentTime-lastTime);
	if(GLView.fps>=33)
	{
		try {
			sleep((int)(GLView.fps-33));
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
	}
}
```

Código 32: Acotar el número de fps

CAPÍTULO 6.- TRABAJOS FUTUROS Y CONCLUSIONES

6.1.- Trabajos futuros

Para llevar a cabo un motor gráfico de videojuegos plataformas en 3D para la plataforma Android son necesarios multitud de pasos. Muchos de ellos no se han podido incluir en esta versión del proyecto debido a la falta de tiempo y a la necesidad de establecer y de cumplir unos requisitos mínimos. Por este motivo se enumeran a continuación mejoras que pueden completar en gran medida lo ya implementado.

- La construcción de una mayor variedad de figuras geométricas.

- La mejora en el uso de fondos de tal manera que el motor gráfico proporcione al usuario la capacidad de crear un fondo de grandes dimensiones sin que se vea afectado el rendimiento de la aplicación ni la memoria del dispositivo móvil. Haciendo para ello, uso de técnicas de compresión de texturas.

- Hacer uso de técnicas más avanzadas para el manejo de texturas como el uso del mip mapping (cambio de resolución de texturas a medida que el observador se acerca o aleja de un objeto), bump mapping (efecto visual que permite ver objetos lisos en relieve) o environment mapping (técnica que permite mapear texturas cuyas coordenadas son dependientes de la vista).

- Mejorar el uso de sprites, permitiendo al usuario representar al personaje principal y a los enemigos con la cantidad de imágenes que el desee, sin necesidad de tener que usar 12 imágenes con un tamaño fijo.

- La adición de un módulo de inteligencia artificial que permita al usuario crear enemigos que persigan al personaje principal por los distintos niveles del juego, de tal manera que puedan avanzar en todas las direcciones posibles y evaluar los caminos óptimos para encontrarle con mayor rapidez.

- Hacer uso del motor gráfico creando un videojuego de plataformas en 3D.

6.2.- Conclusiones

Una vez concluido el desarrollo del presente proyecto, se procederá a hacer un balance del resultado final obtenido. Para ello se repasará de manera breve los objetivos personales así como los requisitos técnicos propuestos en el inicio del mismo y se hará una comparación con los resultados finalmente obtenidos.

En cuanto a los objetivos personales se puede afirmar que se han cumplido de manera satisfactoria. A lo largo del proyecto se ha conseguido adquirir un amplio conocimiento de las herramientas necesarias para su construcción.

Entre dichas herramientas cabe destacar OpenGL y más concretamente la especificación OpenGL ES 1.0, de la cual se han adquirido conocimientos acerca de la construcción de figuras sencillas, del manejo avanzado de texturas, del uso de colores e iluminación de la escena y del manejo básico de la cámara y de sus perspectivas.

También se ha conseguido ampliar en gran medida el manejo de la plataforma Android, proporcionando un amplio conocimiento de su funcionamiento interno y de la gestión de recursos y de memoria, así como del entorno de desarrollo avanzado Eclipse, aprendiendo nuevas técnicas para la depuración del código y la validación de los requisitos técnicos mínimos necesarios para el correcto funcionamiento de la aplicación.

En cuanto a los requisitos técnicos se puede afirmar que la mayor parte de ellos se han cumplido de manera satisfactoria, aunque existen ciertos puntos que podrían mejorarse. Con el uso de este motor se pudo llevar a cabo un videojuego de plataformas 3D en Android, sencillo, sin fallos y que permitía al jugador interactuar con la aplicación de una manera agradable. Esto demuestra de manera práctica la viabilidad de dicho motor.

Por último, es necesario mencionar que la implementación de este proyecto no se podría haber llevado a cabo sin el enorme esfuerzo realizando durante los cinco últimos años, en los que cursé la carrera de Ingeniería Técnica de Telecomunicaciones, y que me han permitido adquirir grandes conocimientos tanto técnicos como personales.

ANEXOS

A.- Planificación

A.1.- Planificación inicial

En este apartado se incluye la planificación inicial previa al desarrollo del proyecto. La fecha de comienzo es el 11 de octubre de 2010 y la fecha fin el jueves 2 de junio de 2011, estimando una duración total de 530 horas distribuidas en 8 meses. A continuación se muestran una tabla con las tareas y subtareas realizadas para el desarrollo del mismo, así como la duración estimada de cada una de ellas, la fecha de comienzo y de fin y las tareas o subtareas de las que dependen.

Nº	Nombre de tarea	Duración	Comienzo	Fin	Pred
1	Documentación inicial	48 horas	lun 11/10/10	vie 05/11/10	
2	Instalación entorno de programación	4 horas	lun 11/10/10	mar 12/10/10	
3	Programas de prueba	50 horas	lun 08/11/10	lun 06/12/10	2;1
4	Análisis y diseño	20 horas	mar 07/12/10	vie 17/12/10	3
5	**Implementación del motor gráfico**	**53 días**	**vie 17/12/10**	**vie 20/05/11**	**4**
6	**Dibujar figuras**	**37 días**	**vie 17/12/10**	**jue 31/03/11**	**4**
7	**Diseño de figuras**	**17 días**	**vie 17/12/10**	**vie 11/02/11**	**4**
8	Documentación de figuras	15 horas	vie 17/12/10	lun 27/12/10	4
9	Diseño de un triángulo	5 horas	lun 27/12/10	jue 30/12/10	8
10	Diseño de un cuadrado	7 horas	jue 30/12/10	jue 20/01/11	9
11	Diseño de una pirámide	10 horas	jue 20/01/11	mar 25/01/11	10
12	Diseño de un cubo	12 horas	mar 25/01/11	mar 01/02/11	11
13	Diseño de una esfera	18 horas	mar 01/02/11	vie 11/02/11	12
14	**Iluminación de figuras**	**5 días**	**vie 11/02/11**	**jue 24/02/11**	**13**
15	Documentación iluminación	8 horas	vie 11/02/11	jue 17/02/11	13
16	Configuración de las fuentes de luz	6 horas	jue 17/02/11	vie 18/02/11	15
17	Iluminación de figuras	6 horas	lun 21/02/11	jue 24/02/11	16

18	**Colorear figuras**	**7 días**	**jue 24/02/11**	**vie 11/03/11**	**17**
19	Documentación color	12 horas	jue 24/02/11	jue 03/03/11	17
20	Colorear triangulo	2 horas	jue 03/03/11	vie 04/03/11	19
21	Colorear cuadrado	2 horas	vie 04/03/11	vie 04/03/11	20
22	Colorear pirámide	3 horas	vie 04/03/11	lun 07/03/11	21
23	Colorear cubo	4 horas	lun 07/03/11	mar 08/03/11	22
24	Colorear esfera	5 horas	jue 10/03/11	vie 11/03/11	23
25	**Texturización de figuras**	**8 días**	**vie 11/03/11**	**jue 31/03/11**	**24**
26	Documentación de texturas	14 horas	vie 11/03/11	vie 18/03/11	24
27	Añadir textura a un triángulo	2 horas	lun 21/03/11	lun 21/03/11	26
28	Añadir textura a un cuadrado	3 horas	mar 22/03/11	mar 22/03/11	27
29	Añadir textura a una pirámide	7 horas	jue 24/03/11	vie 25/03/11	28
30	Añadir textura a un cubo	6 horas	lun 28/03/11	jue 31/03/11	29
31	**Implementación partes del juego**	**12 días**	**jue 31/03/11**	**jue 28/04/11**	**30**
32	Dibujar escenario	16 horas	jue 31/03/11	vie 08/04/11	30
33	Dibujar personaje	16 horas	vie 08/04/11	lun 18/04/11	32
34	Dibujar enemigos	16 horas	mar 19/04/11	jue 28/04/11	33
35	**Análisis de jugabilidad**	**16 horas**	**jue 28/04/11**	**jue 19/05/11**	**34**
36	**Memoria**	**71 días**	**lun 08/11/10**	**vie 20/05/11**	**1**
37	**Motivación**	**4 horas**	**lun 08/11/10**	**mar 09/11/10**	**1**
38	**Estado del arte**	**40 horas**	**mar 09/11/10**	**vie 03/12/10**	**37**
39	**Análisis de los casos de uso**	**7 horas**	**vie 17/12/10**	**mar 21/12/10**	**38;4**
40	**Diagrama de secuencia**	**8 horas**	**jue 23/12/10**	**lun 27/12/10**	**39**
41	**Requisitos**	**40 días**	**lun 27/12/10**	**vie 15/04/11**	**40**
42	Análisis de los requisitos de usuario	10 horas	lun 27/12/10	vie 31/12/10	40
43	Análisis de los requisitos software	14 horas	vie 31/12/10	jue 27/01/11	42
44	**Implementación**	**44,75 d**	**jue 27/01/11**	**lun 23/05/11**	**43**
45	Diseño y construcción de figuras	20 horas	jue 27/01/11	lun 07/02/11	8;43
46	Iluminación y materiales	10 horas	jue 17/02/11	mar 22/02/11	15;45
47	Color	10 horas	jue 03/03/11	mar 08/03/11	19;46
48	Texturas	20 horas	lun 21/03/11	vie 01/04/11	47;26
49	Escenario del videojuego	9 horas	mar 03/05/11	vie 20/05/11	51;32

50	Personaje principal del videojuego	9 horas	mar 19/04/11	vie 22/04/11	48;33
51	Enemigos del videojuego	9 horas	jue 28/04/11	mar 03/05/11	50;34
52	**Pruebas**	**3 horas**	**vie 20/05/11**	**vie 20/05/11**	**35;51**
53	**Anexos**	**4 horas**	**vie 20/05/11**	**lun 23/05/11**	**52**
54	Validación	20horas	mar 24/05/11	vie 03/06/11	35;53

Tabla 4: Planificación inicial

A.1.1 Diagrama de Gantt

El diagrama de Gantt sitúa de forma secuencial las diferentes tareas del proyecto a lo largo de una línea temporal que representa el tiempo total del mismo.

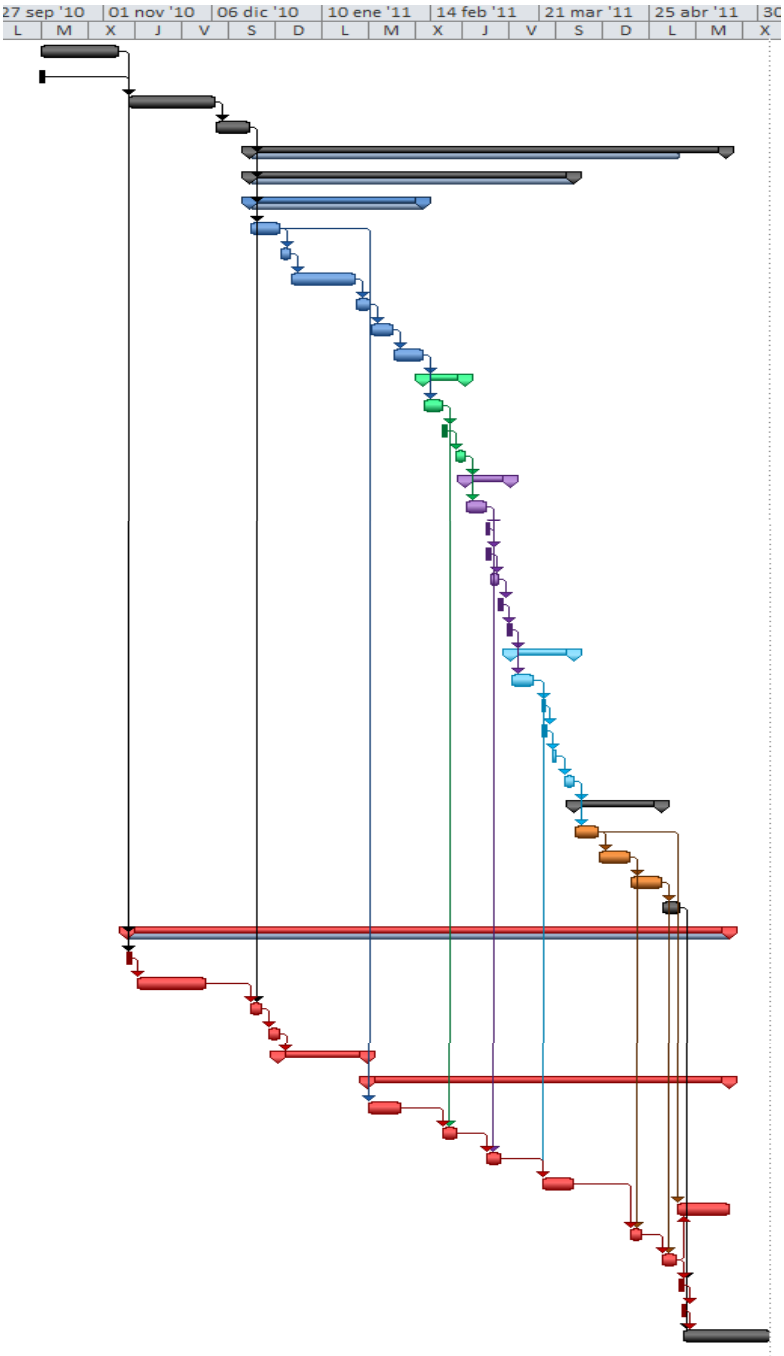


Figura 78: Diagrama de Gantt planificación inicial

A.2.- Planificación final

La fecha de finalización prevista en la planificación inicial, no se corresponde con la fecha real de fin de proyecto por las siguientes causas:

- Necesidad de realizar distintos planteamientos hasta alcanzar una idea innovadora, original y con la suficiente complejidad para el desarrollo de un proyecto fin de carrera, para dos alumnos, que cumpliera con los requisitos fijados por el tutor.
- Dificultad para encontrar información acerca del manejo de OpenGL ES 1.0 al ser una herramienta bastante novedosa.
- Aprendizaje de las técnicas y programas necesarios para el desarrollo de un videojuego de plataformas en 3D para Android.
- El objetivo fijado en el inicio del proyecto no permitía desarrollar dos proyectos con entidad suficiente, por lo que fue necesario realizar una serie de ajustes, definiendo otro objetivo para que los dos alumnos dispusieran de un proyecto propio.

A continuación se muestra una tabla con las tareas más significativas que se retrasaron, la desviación en el tiempo con respecto de la planificación inicial y los motivos que produjeron este retraso.

Nombre de tarea	Duración planificada	Duración real	Variación	Causas
Documentación inicial	20 horas	48 horas	28 horas	▪ Dificultad para encontrar documentación de la herramienta OpenGL ES.
Instalación entorno	4 horas	6 horas	2 horas	▪ Incremento del tiempo de descarga.
Análisis y diseño	20 horas	50 horas	30 horas	▪ Dificultad para encontrar una idea innovadora y con la suficiente complejidad. ▪ Realizar una serie de ajustes para dividir el proyecto inicial en dos proyectos con entidad suficiente.

Nombre de tarea	Duración planificada	Duración real	Variación	Causas
Dibujar figuras	147 horas	232 horas	85 horas	▪ Dificultad para obtener información acerca de la iluminación, el color y la texturización en OpenGL ES. ▪ Retraso en la construcción de figuras debido a la dificultad para unir los vértices que forman las mismas. ▪ Dificultad para añadir texturas en la superficie de las figuras.
Implementación partes del juego	48 horas	84 horas	36 horas	▪ Dificultad para dividir el proyecto inicial. ▪ Dificultad para añadir los sprites de los personajes principales y de los enemigos, de manera que el número de fps no se viera alterado en gran medida. ▪ Dificultad para crear el escenario de manera que ni los fps ni y la memoria interna del dispositivo se vieran alterados.
Análisis de jugabilidad	16 horas	18 horas	2 horas	▪ Las pruebas de jugabilidad del juego se tuvieron que repetir debido al elevado número de fps.
Memoria	177 horas	250 horas	73 horas	▪ Dificultades para obtener información acerca de los motores gráficos en 3D y de OpenGL ES. ▪ Cambio de los requisitos.
Validación	20 horas	50 horas	30 horas	▪ Retraso en la corrección del proyecto final. ▪ Mejora de algunos aspectos.

Tabla 5: Comparativa de la planificación inicial con la planificación final

A continuación se muestra una tabla con la planificación final del mismo. La fecha de inicio corresponde con el 11 de octubre de 2010 y la fecha fin con el 3 de octubre del 2011, siendo necesario para el desarrollo del proyecto un total de 788 horas, distribuidas en 1 año.

Nº	Nombre de tarea	Duración	Comienzo	Fin	Pred.
1	Documentación inicial	48 horas	lun 11/10/10	vie 05/11/10	
2	Instalación entorno de programación	6 horas	lun 11/10/10	mar 12/10/10	
3	Programas de prueba	50 horas	lun 12/11/10	lun 06/12/10	2;1
4	Análisis y diseño	50 horas	mar 07/12/10	vie 21/01/11	3
5	**Implementación del motor gráfico**	**81 días**	**vie 21/01/11**	**jue 11/08/11**	**4**
6	**Dibujar figuras**	**58 días**	**vie 21/01/11**	**vie 17/06/11**	**4**
7	**Diseño de figuras**	**27 días**	**vie 21/01/11**	**vie 25/03/11**	**4**
8	Documentación de figuras	24 horas	vie 21/01/11	vie 04/02/11	4
9	Diseño de un triángulo	8 horas	vie 04/02/11	mar 08/02/11	8
10	Diseño de un cuadrado	12 horas	jue 10/02/11	mar 15/02/11	9
11	Diseño de una pirámide	18 horas	jue 17/02/11	vie 25/02/11	10
12	Diseño de un cubo	21 horas	vie 25/02/11	vie 11/03/11	11
13	Diseño de una esfera	25 horas	vie 11/03/11	vie 25/03/11	12
14	**Iluminación de figuras**	**6 días**	**vie 25/03/11**	**vie 08/04/11**	**13**
15	Documentación iluminación	10 horas	vie 25/03/11	jue 31/03/11	13
16	Configuración de las fuentes luz	7 horas	vie 01/04/11	lun 04/04/11	15
17	Iluminación de figuras	7 horas	mar 05/04/11	vie 08/04/11	16
18	**Colorear figuras**	**11 días**	**vie 08/04/11**	**mar 03/05/11**	**17**
19	Documentación color	17 horas	vie 08/04/11	lun 18/04/11	17
20	Colorear triangulo	4 horas	mar 19/04/11	jue 21/04/11	19
21	Colorear cuadrado	4 horas	jue 21/04/11	vie 22/04/11	20
22	Colorear pirámide	6 horas	vie 22/04/11	mar 26/04/11	21
23	Colorear cubo	6 horas	mar 26/04/11	vie 29/04/11	22
24	Colorear esfera	7 horas	vie 29/04/11	mar 03/05/11	23
25	**Texturización de figuras**	**14 días**	**mar 17/05/11**	**vie 17/06/11**	**24**
26	Documentación de texturas	23 horas	mar 17/05/11	lun 30/05/11	24
27	Añadir textura a un triángulo	5 horas	lun 30/05/11	jue 02/06/11	26

28	Añadir textura a un cuadrado	6 horas	jue 02/06/11	vie 03/06/11	27
29	Añadir textura a una pirámide	10 horas	lun 06/06/11	vie 10/06/11	28
30	Añadir textura a un cubo	12 horas	vie 10/06/11	vie 17/06/11	29
31	**Implementación partes del juego**	**21 días**	**vie 17/06/11**	**vie 05/08/11**	**30**
32	Dibujar escenario	30 horas	vie 17/06/11	mar 05/07/11	30
33	Dibujar personaje	27 horas	mar 05/07/11	jue 21/07/11	32
34	Dibujar enemigos	27 horas	vie 22/07/11	vie 05/08/11	33
35	**Análisis de jugabilidad**	18 horas	vie 05/08/11	jue 11/08/11	34
36	**Memoria**	**117 días**	**lun 08/11/10**	**lun 05/09/11**	**1**
37	**Motivación**	**16 horas**	**lun 08/11/10**	**mar 16/11/10**	**1**
38	**Estado del arte**	**64 horas**	**mar 16/11/10**	**vie 24/12/10**	**37**
39	**Análisis de los casos de uso**	**12 horas**	**vie 21/01/11**	**vie 28/01/11**	**38;4**
40	**Diagrama de secuencia**	**11 horas**	**vie 28/01/11**	**vie 04/02/11**	**39**
41	**Requisitos**	**20 días**	**vie 04/02/11**	**vie 18/02/11**	**40**
42	Análisis de los requisitos usuario	10 horas	vie 18/02/11	jue 10/02/11	40
43	Análisis de los requisitos software	14 horas	jue 10/02/11	vie 18/02/11	42
44	**Implementación**	**72 días**	**vie 18/02/11**	**mar 16/08/11**	**43**
45	Diseño y construcción de figuras	24 horas	vie 18/02/11	vie 04/03/11	8;43
46	Iluminación y materiales	13 horas	vie 01/04/11	vie 08/04/11	15;45
47	Color	13 horas	mar 19/04/11	mar 26/04/11	19;46
48	Texturas	25 horas	lun 30/05/11	lun 13/06/11	47;26
49	Escenario del videojuego	12 horas	jue 11/08/11	mar 16/08/11	51;32
50	Personaje principal del videojuego	12 horas	vie 22/07/11	jue 28/07/11	48;33
51	Enemigos del videojuego	12 horas	vie 05/08/11	mié 10/08/11	50;34
52	**Pruebas**	**5 horas**	**jue 01/09/11**	**vie 02/09/11**	**35;51**
53	**Anexos**	**7 horas**	**vie 02/09/11**	**lun 05/09/11**	**52**
54	Validación	50 horas	lun 05/09/11	lun 03/10/11	35;53

Tabla 6: Planificación final

A.2.1.- Diagrama de Gantt

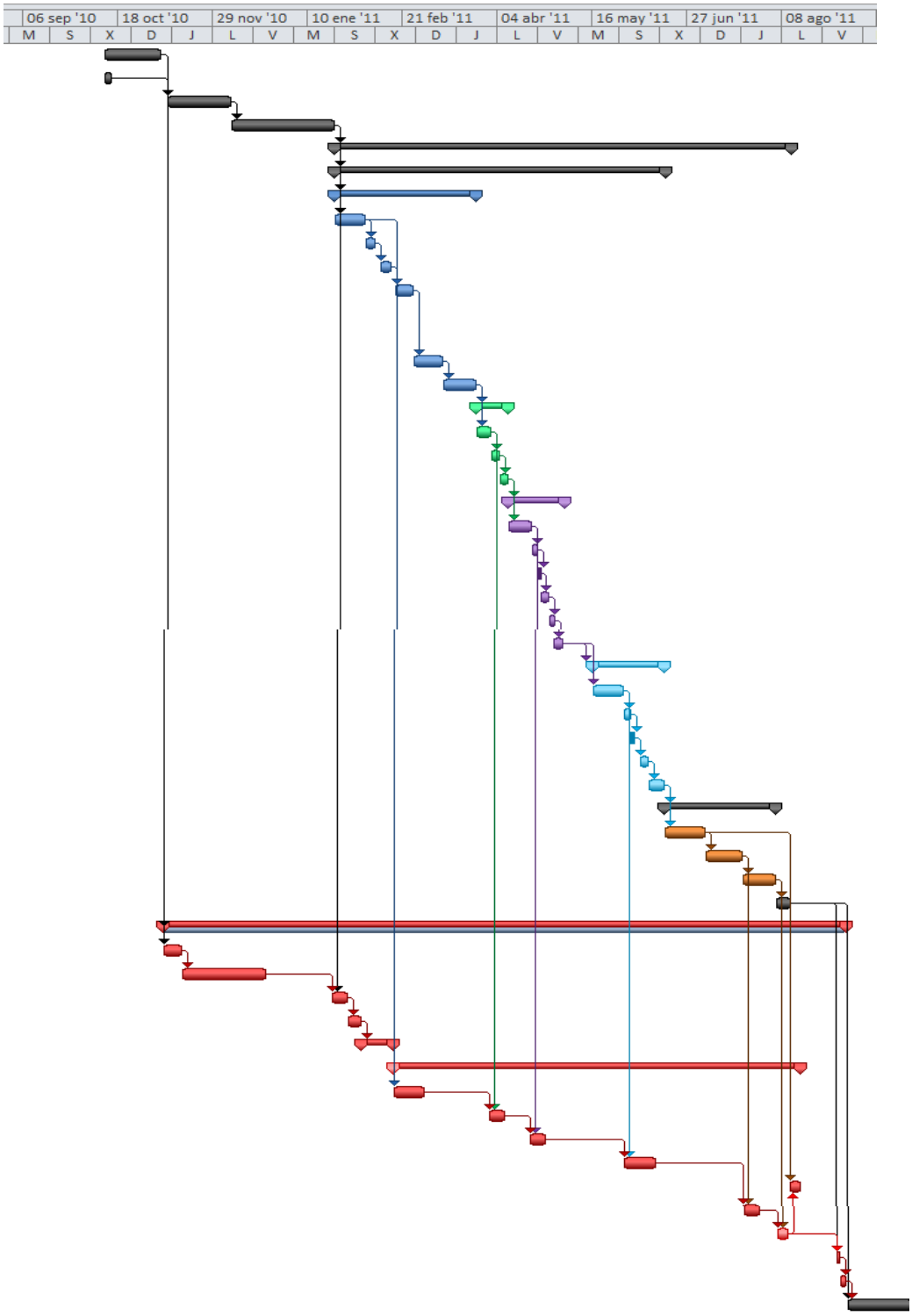


Figura 79: Diagrama de Gantt planificación final

B.- Presupuesto

El presupuesto se desglosa en costes de material y costes de personal según se detalla en los siguientes apartados.

B.1.- Costes de equipamiento informático

Los materiales empleados para la realización del proyecto fueron los siguientes:

Concepto	Características
Ordenador portátil	Fabricante: Hewlett – Packard Modelo: Compaq Presario C700 Notebook PC Procesador: Intel ® Core ™ 2 Duo CPU T5750 Memoria RAM: 3,00 GB
Teléfono móvil	Modelo: HTC Wildfire Memoria ROM: 512 Mb Memoria RAM: 384 Mb Velocidad CPU: 528 Mhz Sistema Operativo: Android™ 2.1 (Eclair)

Tabla 7: Materiales empleados

A continuación se detalla el coste de equipamiento teniendo en cuenta el precio, el periodo de dedicación y el periodo de amortización de los mismos.

Concepto	Precio	Periodo dedicación	Periodo amortización	Coste amortización
Compaq Presario C700 Notebook PC	499,00 €	12 Meses	60 Meses	99,8 €
Teléfono móvil modelo HTC Wildfire	239,00 €	11 Meses	12 Meses	219,08€
TOTAL	**638,00€**			**318,88 €**

Tabla 8: Costes finales de equipamiento

El coste total de material estimado se muestra en la siguiente tabla.

Concepto	Precio	Periodo dedicación	Periodo amortización	Coste amortización
Compaq Presario C700 Notebook PC	499,00 €	6 Meses	60 Meses	49,9 €
Teléfono móvil modelo HTC Wildfire	239,00 €	5 Meses	12 Meses	99,18€
TOTAL	**638,00€**			**149,08€**

Tabla 9: Costes estimados de equipamiento

Hay una desviación de 169,8€ en el coste de material.

B.2.- Costes de personal

El proyecto ha sido llevado a cabo por un analista programador y un jefe de proyecto.

Cargo	Coste hora
Analista Programador	35,00€
Jefe de Proyecto	45,00€

Tabla 10: Personal del proyecto

El coste total de personal se calcula mediante el producto del coste hora de cada participante por el total de horas realizadas por el mismo.

Cargo	Coste hora	Horas	Importe
Programador	30,00€	718	21.540,00 €
Jefe de Proyecto	40,00€	70	2.800.00 €
TOTAL		788	24.340,00 €

Tabla 11: Coste final de personal

El coste total de personal estimado se muestra en la siguiente tabla.

Cargo	Coste hora	Horas	Importe
Programador	30,00€	495	14.850,00 €
Jefe de Proyecto	40,00€	35	1.400.00 €
TOTAL		530	16.250,00 €

Tabla 12: Coste estimado de personal

Hay una desviación de 8.090€ en el coste de personal.

B.3.- Coste Total

En la siguiente tabla se desglosa el balance final del coste del proyecto:

Concepto	Precio
Costes materiales	318,88 €
Costes de Personal	24.340,00 €
Subtotal	24.658,88 €
I.V.A. (18%)	4.438,60 €
TOTAL	**29.097,48 €**

Tabla 13: Coste final

En la siguiente tabla se desglosa el balance final del coste estimado del proyecto:

Concepto	Precio
Costes materiales	149,08 €
Costes de Personal	16.250,00 €
Subtotal	16.399,08 €
I.V.A. (18%)	2.951,83 €
TOTAL	**19.350,91 €**

Tabla 14: Coste final estimado

Hay una desviación de 9746,57€ en el coste total del proyecto.

El presupuesto total de este proyecto asciende a la cantidad de **29.097,48 EUROS.**

C.- Descripción del videojuego

Cube´s War es un videojuego de plataformas en 3D, implementado por un alumno de la Universidad Carlos III, que se llevó a cabo para probar el motor gráfico diseñado en este proyecto fin de carrera.

El juego se desarrolla en un escenario tridimensional, donde el usuario controla el movimiento del personaje mediante un pad situado en la parte inferior de la pantalla. Dicho escenario permite al personaje no sólo desplazarse horizontal y verticalmente, sino que también le permite avanzar o retroceder en profundidad.

El objetivo del juego es avanzar de un escenario a otro, superando los distintos obstáculos que se le presentan. Para conseguirlo el personaje deberá obtener un objeto, en este caso una llave, que le permita abrir una puerta situada al final de cada nivel. El juego finaliza cuando el usuario consigue pasar tres niveles distintos, cada uno de ellos ambientado en un paisaje diferente.

Entre los distintos obstáculos que el usuario puede encontrar a lo largo del videojuego se destacan los enemigos. Hay varios tipos de enemigos que se diferencian por su apariencia, por la velocidad a la que se aproximan al personaje y por el movimiento que realizan en los distintos ejes de coordenadas.

Otro elemento del juego que facilita al usuario el logro del objetivo final es el cambio de perspectiva del escenario mediante el sensor de movimiento que disponen los distintos dispositivos móviles Android.

En todo momento el usuario podrá pausar la partida y retroceder al menú principal. Dicho menú proporcionará entre otras, la posibilidad de reiniciar el juego, configurar los efectos de sonido y de vibración del dispositivo móvil y salir de la aplicación.

GLOSARIO

- **A**

[1] **API (Application Programming Interface):** El interfaz de programación de aplicaciones o API es un conjunto funciones y procedimientos que ofrece una biblioteca para ser utilizado por otro software como una capa de abstracción. [↑]

[2] **Array:** colección **ordenada** de elementos de un mismo tipo de datos, agrupados de forma consecutiva en memoria. Cada elemento del array tiene asociado un índice, que no es más que un número natural que lo identifica inequívocamente y permite al programador acceder a él. [↑]

- **B**

[3] **Biblioteca:** agrupación de código que proporcionan servicios a programas independientes pasando a formar parte de éstos. Permiten la distribución de funcionalidades y la construcción modular. También conocido como librería. [↑]

[4] **BSP:** Estructura de datos utilizada en computación gráfica para reducir el número de polígonos de una escena y así, poder realizar más rápidamente su despliegue. Esta técnica consiste en dividir recursivamente un nivel en dos subniveles hasta satisfacer uno o más requisitos, formando un árbol binario donde cada nodo representa un nivel o área particular del espacio. [↑]

- **C**

[5] **C:** lenguaje de programación, orientado a la implementación de Sistemas Operativos, creado en 1972 por Dennis M. Ritchie en los Laboratorios Bell como evolución del anterior lenguaje B. [↑]

[6] **C++:** lenguaje de programación diseñado a mediados de los años 1980 por Bjarne Stroustrup. La intención de su creación fue el extender al exitoso lenguaje de programación C con mecanismos que permitan la manipulación de objetos. [↑]

[7] **Computador PDP-1:** primer computador en serie PDP de la Digital Equipment, producida por primera vez en 1960. [↑]

- **D**

[8] **Dalvik:** Nombre de la máquina virtual utilizada por Android. Está específicamente adaptada a las características de rendimiento de un dispositivo móvil. Trabaja con ficheros .dex que obtiene a partir del bytecode de Java. [↑]

[9] **DirectX:** colección de API desarrolladas para facilitar las complejas tareas relacionadas con multimedia, especialmente programación de juegos y vídeo, en la plataforma Microsoft Windows. [↑]

- **E**

[10] **ESDAC (Electronic Delay Storage Automatic Calculator):** uno de los primeros computadores en tener un programa almacenado en memoria y que, en 1951, introdujo el concepto de microprogramación al hacer que la CPU del computador estuviese controlada por un programa escrito la ROM del sistema. [↑]

- **F**

[11] **FPS (First Person Shooter) o Videojuego de disparos en primera persona:** es un género de videojuegos y subgénero de los videojuegos de disparos que se desarrolla desde la perspectiva del personaje protagonista y en la que, típicamente, sólo se ven las manos del personaje. [↑]

[12] **Física ragdoll:** En el motor físico de una computadora, la física ragdoll o pelele es un tipo de proceso de animación, con frecuencia usado como reemplazo de las animaciones estáticas de muerte tradicionales. [↑]

[13] **frame (fotograma o cuadro):** imagen particular dentro de una sucesión de imágenes que componen una animación. La continua sucesión de estos fotogramas producen a la vista la sensación de movimiento, fenómeno dado por las pequeñas diferencias que hay entre cada uno de ellos. [↑]

[14] **fps (frames por segundo):** unidad de medida que expresa el número de fotogramas por segundo que se necesitan para crear movimiento. [↑]

[15] **FPU:** es un componente de la unidad central de procesamiento especializado en el cálculo de operaciones en coma flotante. Las operaciones básicas que toda FPU puede realizar son la suma y multiplicación usuales, aunque hay algunos sistemas capaces de realizar cálculos trigonométricos o exponenciales. [↑]

[16] **Framework:** estructura conceptual y tecnológica de soporte definido, normalmente con artefactos o módulos de software concretos, con base a la cual otro proyecto de software puede ser más fácilmente organizado y desarrollado. Típicamente, puede incluir soporte de programas, bibliotecas, y un lenguaje interpretado, entre otras herramientas, para así ayudar a desarrollar y unir los diferentes componentes de un proyecto. [↑]

- **G**

[17] **GNU/Linux:** es uno de los términos empleados para referirse a la combinación del núcleo o kernel libre similar a Unix denominado Linux, que es usado con herramientas de sistema GNU. Su desarrollo es uno de los ejemplos más prominentes de software libre; todo su código fuente puede ser utilizado, modificado y redistribuido libremente por cualquiera bajo los términos de la GPL (Licencia Pública General de GNU, en inglés: General Public License) y otra serie de licencias libres. [↑]

[18] **Gráficos EGA:** es el acrónimo inglés de Enhanced Graphics Adapter, la especificación estándar de IBM PC para visualización de gráficos, situada entre CGA y VGA en términos de rendimiento gráfico (es decir, amplitud de colores y resolución). [↑]

- **H**

[19] **Hilo/ Thread:** En sistemas operativos, un hilo o thread constituye cada uno de los flujos de ejecución en el que puede ser divido un proceso. Todos los hilos de un proceso comparten espacio en memoria o variables globales. Permiten la ejecución concurrente de varias tareas. [↑]

- **I**

[20] **IBM PC:** es la versión original y el progenitor de la plataforma de hardware compatible IBM PC. Es el IBM modelo 5150, y fue introducido el 12 de agosto de 1981. Fue creado por un equipo de ingenieros y de diseñadores bajo la dirección de Don Estridge del IBM Entry Systems Division en Boca Raton, Florida. [↑]

[21] **Isométrica:** Constituye una representación visual de un objeto tridimensional en dos dimensiones, en la que los tres ejes ortogonales principales, al proyectarse, forman ángulos de 120º, y las dimensiones paralelas a dichos ejes se miden en una misma escala. [↑]

- **J**

[22] **Java:** es un lenguaje de programación orientado a objetos, desarrollado por Sun Microsystems a principios de los años 90. El lenguaje en sí mismo toma mucha de su sintaxis de C y C++, pero tiene un modelo de objetos más simple y elimina herramientas de bajo nivel, que suelen inducir a muchos errores, como la manipulación directa de punteros o memoria. Con respecto a la memoria, su gestión no es un problema ya que ésta es gestionada por el propio lenguaje y no por el programador. [↑]

- **K**

[23] **Kernel:** Núcleo de un sistema operativo y por lo tanto, pieza clave para el funcionamiento del mismo. Responsable de dar acceso al hardware, gestionar recursos y hacer llamadas al sistema. [↑]

- **L**

[24] **LittleBigPlanet:** es un videojuego de plataformas y de lógica desarrollado por Media Molecule, una compañía británica, y publicado por Sony Computer Entertainment Europe.

En él, los jugadores tendrán que controlar por medio de las palancas analógicas a pequeños personajes llamados Sackboy que tienen la habilidad de correr, saltar, colgarse, arrastrar y empujar objetos. Además del movimiento regular izquierdo-y-derecho, y a pesar de la mirada 2D del juego, los niveles consisten en tres niveles de profundidad - el primer plano, el medio y el fondo - y éstos pueden ser atravesados entre sí automáticamente por el mismo juego, o por orden del jugador. [↑]

- **M**

[25] **Middleware:** Capa de abstracción del software que posibilita el funcionamiento de aplicaciones distribuidas sobre plataformas heterogéneas. [↑]

[26] **Mapeado de texturas:** Consiste en aplicar la textura directamente sobre el objeto, como si se tratase de un plano. Es ideal para suelos, paredes y demás superficies lisas, y consecuentemente, poco apropiado para superfices no planas. [↑]

[27] **Motor gráfico:** es la parte de un programa que controla, gestiona y actualiza los gráficos 3D en tiempo real. [↑]

- **P**

[28] **Pentaminós:** Un pentominó es una poliforma de la clase poliominó que consiste en una figura geométrica compuesta por cinco cuadrados unidos por sus lados. Existen doce pentominós diferentes, que se nombran con diferentes letras del abecedario. Alekséi Pázhitnov se inspiró en ellos al crear el videojuego Tetris. [↑]

[29] **Pipeline gráfica:** consiste en múltiples unidades aritméticas o CPUs completas, que implementan variados escenarios de operaciones típicas de renderizado, por ejemplo, cálculos de luz y colores, renderizado, proyección de perspectiva, etc. [↑]

- **R**

[30] **Renderizado:** es un término usado en jerga informática para referirse al proceso de generar una imagen desde un modelo. En términos de visualizaciones en una computadora, más específicamente en 3D, la renderización es un proceso de cálculo complejo desarrollado por un ordenador destinado a generar una imagen 2D a partir de una escena 3D. [↑]

- **S**

[31] **Smartphones:** Dispositivo móvil que representa una evolución de los teléfonos móviles con la inclusión de pantalla táctil, teclado, conexión Wi-Fi, aplicaciones de usuario como navegador web o cliente de correo. [↑]

[32] **Sprites:** tipo de mapa de bits dibujados en la pantalla de ordenador por hardware gráfico especializado sin cálculos adicionales de la CPU. A menudo son pequeños y parcialmente transparentes y se usan en videojuegos para crear los gráficos de los protagonistas. [↑]

[33] **Survival horror**: Conocido también como "terror y *supervivenciar"* es un género de videojuegos que utiliza distintos elementos para crear una atmósfera de terror psicológico en el jugador. Destacan por la poca libertad de movimientos (caminar, correr, apuntar y atacar) y la escasez de recursos, por presencia de puzzles o acertijos y por requerir una capacidad de investigación y observación detallada por parte del jugador. [↑]

- **T**

[34] **Texturas:** es una imagen del tipo bitmap utilizada para cubrir la superficie de un objeto virtual, ya sea tridimensional o bidimensional, con un programa de gráficos especial. Multitexturizado es el uso de más de una textura a la vez en un polígono. Más en profundidad es un conjunto de primitivas o elementos denominados "texels", estos se asimilan a un conjunto contiguo de elementos (pixeles en 2D) con alguna propiedad tonal o regional. [↑]

- **W**

[35] **WebKit:** es una plataforma para aplicaciones que funciona como base para el navegador web Safari, Google Chrome, Epiphany, Maxthon, Midori entre otros. Está basado originalmente en el motor de renderizado KHTML del navegador web del proyecto KDE, Konqueror. [↑]

- **Z**

[36] **Z-buffering:** Es una parte de la memoria gráfica reservada a almacenar los valores del eje z para cada pixel. El algoritmo z buffer permite descartar partes de los objetos que se cubren entre sí. [↑]

REFERENCIAS

[1] Retro Informática, “Historia de los videojuegos”. Disponible en World Wide Web: <http://www.fib.upc.edu/retro-informatica/historia/videojocs.html>. [Último acceso Junio 2011].

[2] El otro lado, “Historia de los videojuegos: Los inicios”. Disponible en World Wide Web: < http://www.elotrolado.net/wiki/Historia_de_los_videojuegos:_Los_inicios>. [Último acceso Junio 2011].

[3] JJ Velasco, redactor de ALT1040, “Historia de la tecnología: OXO, un videojuego para uno de los primeros computadores de la historia”. Disponible en World Wide Web: <http://alt1040.com/2011/07/oxo-un-videojuego-para-uno-de-los-primeros-computadores-de-la-historia>. 15 de Julio 2011. [Último acceso Julio 2011].

[4] JJ Velasco, redactor de ALT1040, “Tennis for Two, uno de los primeros videojuegos de la historia”. Disponible en World Wide Web: <http://alt1040.com/2011/07/tennis-for-two-uno-de-los-primeros-videojuegos-de-la-historia>. 8 de Julio 2011. [Último acceso Julio 2011].

[5] Computer History Museum, “Spacewar!”. Disponible en World Wide Web: <http://pdp-1.computerhistory.org/pdp-1/?f=theme&s=4&ss=3>. [Último acceso Junio 2011].

[6] Infoconsolas, “Historia de los videojuegos, los inicios”. Disponible en World Wide Web: <http://www.infoconsolas.com/general/historia-de-los-videojuegos-los-inicios>. 1 de Septiembre de 2009. [Último acceso Junio 2011].

[7] JJ Velasco redactor de ALT1040, “Historia de la tecnología: Spacewar!, el videojuego que nació en el MIT”. Disponible en World Wide Web: <http://alt1040.com/2011/07/spacewar-el-videojuego-que-nacio-en-el-mit>. 29 de Julio de 2011. [Último acceso Agosto 2011].

[8] JJ Velasco redactor de ALT1040, “Historia de la tecnología: Galaxy Game, la primera máquina recreativa de la historia”. Disponible en World Wide Web: <http://alt1040.com/2011/08/galaxy-game-primera-maquina-recreativa-de-la-historia>. 5 de Agosto 2011. [Último acceso Junio 2011].

[9] Abadia Digital, “Computer Space, la primera máquina recreativa de la historia”. Disponible en World Wide Web: < http://www.abadiadigital.com/articulo/computer-space-la-primera-maquina-recreativa-de-la-historia/>. 13 de Septiembre de 2009. [Último acceso Junio 2011].

[10] Wikipedia, la enciclopedia libre,”Computer Space”. Disponible en World Wide Web: <http://es.wikipedia.org/wiki/Computer_Space>. [Último acceso Junio 2011].

[11] Scenebeta, “1972-1974 Magnavox Odyssey: la madre de todas las consolas“. Disponible en World Wide Web: < http://www.scenebeta.com/tutorial/1972-1974-magnavox-odyssey-la-madre-de-todas-las-consolas >. 27 Agosto de 2007. [Último acceso Junio 2011].

[12] David Winter, “Magnavox Odyssey - First home video game console” <http://www.pong-story.com/odyssey.htm>. [Último acceso Junio 2011].

[13] Wikipedia, “Modelo de color RGB”. Disponible en World Wide Web: <http://es.wikipedia.org/wiki/Modelo_de_color_RGB>. [Último acceso Octubre 2011].

[14] El otro lado, "Historia de los videojuegos: Los inicios". Disponible en World Wide Web: <http://www.elotrolado.net/wiki/Historia_de_los_videojuegos:_Decada_de_los_70>. [Último acceso Junio 2011].

[15] Fernando José Serrano García, "Blending y masking". Disponible en World Wide Web: <http://usuarios.multimania.es/andromeda_studios/paginas/tutoriales/tutgl008.htm>. [Último acceso Octubre 2011].

[16] Museo del videojuego. "La definitiva historia de Atari". Disponible en World Wide Web: < http://www.museodelvideojuego.com/2007/12/25/la-definitiva-historia-de-atari/>. 25 de Diciembre de 2007. [Último acceso Junio 2011].

[17] Taringa!, "Generaciones de consolas de juegos". Disponible en World Wide Web: <http://www.taringa.net/posts/info/1613484/Generaciones-de-consolas-de-juegos.html>. [Último acceso Junio 2011].

[18] Nintendo, "La historia de Nintendo". Disponible en World Wide Web: http://www.nintendo.es/NOE/es_ES/service/la_historia_de_nintendo_9911.html>. [Último acceso Junio 2011].

[19] Eduardo Rivera, "OpenGL: color, iluminación y materiales". Disponible en World Wide Web: < http://es.scribd.com/doc/7759530/OpenGL-Color-Iluminacion-y-Materiales>. [Último acceso Octubre 2011].

[20] El otro lado, "Historia de los videojuegos: Década de los 80". Disponible en World Wide Web: < http://www.elotrolado.net/wiki/Historia_de_los_videojuegos:_Decada_de_los_80 >. [Último acceso Julio 2011].

[21] El otro lado, "Historia de los videojuegos: Década de los 90". Disponible en World Wide Web: < http://www.elotrolado.net/wiki/Historia_de_los_videojuegos:_Decada_de_los_90>. [Último acceso Julio 2011].

[22] Wikia, "Doom rendering engine". Disponible en World Wide Web: < http://doom.wikia.com/wiki/Doom_rendering_engine >. [Último acceso Julio 2011].

[23] "Iluminación y sombreado en OpenGL". Disponible en World Wide Web: <http://www.fing.edu.uy/inco/cursos/compgraf/Clases/2011/14-Iluminacion%20y%20Sombreado%5BopenGL%5D.pdf>. [Último acceso Octubre 2011].

[24] Ionlitio, "Duke Nukem 3D', el origen de la leyenda". Disponible en World Wide Web: < http://www.ionlitio.com/duke-nukem-3d-el-origen-de-la-leyenda/>. 6 Abril de 2011. [Último acceso Julio 2011].

[25] Rubén Talón Argente, "Motores gráficos". Disponible en World Wide Web: < informatica.uv.es/iiguia/IG/motores_graf.pps>. [Último acceso Agosto 2011].

[26] Nicolás Fernández Marchetti, "Evolución de los motores gráficos". Disponible en World Wide Web: < http://www.slideshare.net/nfmarchetti/evolucin-de-los-motores-grficos-presentation >. [Último acceso Agosto 2011].

[27] Taringa!, "Historia de los motores gráficos de juegos". Disponible en World Wide Web: < http://www.taringa.net/posts/info/5715119/Historia-de-los-Motores-Graficos-de-Juegos.html >. [Último acceso Agosto 2011].

[28] Wikipedia, “Maze War”. Disponible en World Wide Web: < http://es.wikipedia.org/wiki/Maze_War>. [Último acceso Julio 2011].

[29] 3D Juegos, “Todo sobre Nintendo”. Disponible en World Wide Web: <http://www.3djuegos.com/comunidad-foros/tema/8875901/0/la-historia-de-nintendo-desde-sus-inicios-mas-humildes-hasta-su-presente/>. [Último acceso Julio 2011].

[30] Comunidad Guadalupana, “Space Invaders – juego clasico online”. Disponible en World Wide Web: <http://blog.comunidadguadalupana.com/2010/05/space-invaders-juego-clasico-online/>. [Último acceso Julio 2011].

[31] Androideity, “Libgdx para crear juegos en Android”. Disponible en World Wide Web: < http://androideity.com/2011/08/22/libgdx-para-crear-juegos-en-android/>. 22 de Agosto 2011. [Último acceso Agosto 2011].

[32] Androideity, “AndEngine, el motor de juegos 2D OpenGL para Android”. Disponible en World Wide Web: < http://androideity.com/2011/08/23/andengine-el-motor-de-juegos-2d-opengl-para-android/>. [Último acceso Agosto 2011].

[33] Taringa!, “Historia del teléfono celular (A través del tiempo)”. Disponible en World Wide Web: < http://www.taringa.net/posts/info/2176951/Historia-del-telefono-celular-_-A-traves-del-tiempo_.html >. [Último acceso Junio 2011].

[34] ESCET- Universidad Rey Juan Carlos de Madrid, “Iluminación”. Disponible en World Wide Web: <http://dac.escet.urjc.es/docencia/GV3D/DocGL4.pdf>. [Último acceso Octubre 2011].

[35] Neoteo, “La Historia de Atari”. Disponible en World Wide Web: < http://www.neoteo.com/la-historia-de-atari >. [Último acceso Junio 2011].

[36] Wikipedia, “Esfera”. Disponible en World Wide Web: <http://es.wikipedia.org/wiki/Esfera>. [Último acceso Octubre 2011].

[37] Índice Latino, “Historia de los videojuegos: Videoconsolas”. Disponible en World Wide Web: <http://indicelatino.com/juegos/historia/consolas/>. [Último acceso Junio 2011].

[38] Fase Bonus, “Pitfall”. Disponible en World Wide Web: <http://www.fasebonus.net/index.php?option=com_content&view=article&id=567:pitfall&catid=36:2011>. [Último acceso Agosto 2011].

[39] Twikiteros, “La Segunda Generación de las Consolas de Videojuego de 1976 a 1984”. Disponible en World Wide Web: <http://www.twikiteros.com/noticias_curiosas/7782-historia_de_las_consolas_de_videojuego_%5B2da_generaci%F3n%5D.html>. [Último acceso Junio 2011].

[40] Juan García, “La historia de Nintendo a través de la publicidad 1977-1983”. Disponible en World Wide Web: <http://ecetia.com/2009/05/la-historia-de-nintendo-a-traves-de-la-publicidad-1977-1983>. 17 de Mayo de 2009. [Último acceso Junio 2011].

[41] Sonic X, “Historia de los videojuegos”. Disponible en World Wide Web: <http://sonicx2000.tripod.com/id1.html>. [Último acceso Junio 2011].

[42] Ion Litio. Disponible en World Wide Web: <http://www.ionlitio.com/ganadores-del-concurso-half-life-2-the-orange-box/>. [Último acceso Agosto 2011].

[43] Wikipedia, "Build engine". Disponible en World Wide Web: <http://en.wikipedia.org/wiki/Build_engine>. [Último acceso Agosto 2011].

[44] 3D Juegos, "Juegos FPS". Disponible en World Wide Web: <http://www.3djuegos.com/comunidad-foros/tema/836838/0/juegos-fps/>. [Último acceso Agosto 2011].

[45] IdeasGeek, "Videojuegos shooters en primera persona que tienen su sitio en la historia". Disponible en World Wide Web: <http://www.ideasgeek.net/2009/09/19/videojuegos-shooters-en-primera-persona-que-tienen-su-sitio-en-la-historia/>. 19 de Setiembre 2011. [Último acceso Septiembre 2011].

[46] Taringa!, "Unreal Engine: Un Potente Motor Gráfico". Disponible en World Wide Web: <http://www.taringa.net/posts/info/4087260/Unreal-Engine_-Un-Potente-Motor-Grafico.html>. [Último acceso Septiembre 2011].

[47] "Angel Engine". Disponible en World Wide Web: < http://code.google.com/p/angel-engine/>. [Último acceso Agosto 2011].

[48] "JPCT-AE - a free 3D engine for Android". Disponible en Word Wide Web: <http://www.jpct.net/jpct-ae/>. [Último acceso Agosto 2011].

[49] Gonzalo de Córdoba, "Motores de desarrollo para juegos 2D o 3D". Disponible en World Wide Web: <http://www.tutorialandroid.com/avanzado/motores-de-desarrollo-para-juegos-2d-o-3d/>. [Último acceso Agosto 2011].

[50] Against the game, “Texture Mapping – OpenGL Android (Displaying Images using OpenGL and Squares)”. Disponible en World Wide Web: < http://obviam.net/index.php/texture-mapping-opengl-android-displaying-images-using-%20opengl-and-squares/>. 26 Enero 2011. [Último acceso Abril 2011].

[51] Wikipedia, “OpenGL”. Disponible en World Wide Web: <http://es.wikipedia.org/wiki/OpenGL>. [Último acceso Septiembre 2011].

[52] Índice Latino, “Historia de los videojuegos: máquinas recreativas (arcade)”. Disponible en World Wide Web: <http://indicelatino.com/juegos/historia/recreativas/>. [Último acceso Abril 2011].

[53] Wikipedia, “Asteroids”. Disponible en World Wide Web: <http://es.wikipedia.org/wiki/Asteroids>. [Último acceso Septiembre 2011].

[54] LiveMotion, “Museo de Consolas: Coleco Telstar”. Disponible en World Wide Web: <http://livemotion.foros.bz/t3501-museo-de-consolas-coleco-telstar>. [Último acceso Septiembre 2011].

[55] AngelFire, “La historia de los videojuegos”. Disponible en World Wide Web: <http://www.angelfire.com/retro/videojuegos/arcadeh1.htm>. [Último acceso Septiembre 2011].

[56] Wikipedia, “Pac-Man”. Disponible en World Wide Web: <http://es.wikipedia.org/wiki/Pac-Man>. [Último acceso Junio 2011].

[57] Wikipedia, "Nintendo Entertainment System". Disponible en World Wide Web:< http://es.wikipedia.org/wiki/Nintendo_Entertainment_System>. [Último acceso Junio 2011].

[58] Consolas, "Historia complete de Super Mario Bros". Disponible en World Wide Web: <http://www.consolas.es/14-11-2007/generos/historia-completa-de-super-mario-bros>. 14 Noviembre 2007. [Último acceso Junio 2011].

[59] Max Ferzzola, "La historia de Tetris (25 años de Tetris)". Disponible en World Wide Web: <http://www.neoteo.com/la-historia-de-tetris-25-anos-de-tetris-16135>. [Último acceso Junio 2011].

[60] Infoconsolas, "Saga Out Run: ¿Te gusta conducir?". Disponible en World Wide Web: <http://www.infoconsolas.com/general/saga-out-run-%C2%BFte-gusta-conducir>. [Último acceso Junio 2011].

[61] Lisandro Pardo, "La historia de los juegos shareware". Disponible en World Wide Web: <http://www.neoteo.com/la-historia-de-los-juegos-shareware>. [Último acceso Junio 2011].

[62] Wikipedia, "Max Payne (videojuego)". Disponible en World Wide Web: <http://es.wikipedia.org/wiki/Max_Payne_(videojuego)>. [Último acceso Junio 2011].

[63] Taringa!, "Saga Battlefield". Disponible en World Wide Web: <http://www.taringa.net/posts/info/10491225/Saga-Battlefield-_info_.html>. [Último acceso Junio 2011].

[64] ABC Digital, "Juegos de guerra". Disponible en World Wide Web: <http://www.abc.com.py/nota/juegos-de-guerra-parte-1/>. [Último acceso Junio 2011].

[65] Jorge García, "Curso de introducción a OpenGL". Disponible en World Wide Web: <http://www.e-ghost.deusto.es/docs/Manual-opengl.pdf>". [Último acceso Septiembre 2011].

[66] Android Developers. Disponible en World Wide Web: <http://developer.android.com/guide/topics/graphics/opengl.html>". [Último acceso Septiembre 2011].

[67] "OpenGL ES con Android". Disponible en World Wide Web: <http://www.slashmobility.com/joomla/images/stories/presentaciones/openGL/FO-1.OpenGL_ES-OpenGL_ES_Android.pdf>". [Último acceso Septiembre 2011].

[68] edu4 Java, "Android Game Programming 4. Our First Sprites". Disponible en World Wide Web: <http://www.edu4java.com/es/androidgame/androidgame4.html>. [Último acceso Octubre 2011].

[69] "Texturas". Disponible en World Wide Web: <http://sabia.tic.udc.es/gc/Contenidos%20adicionales/trabajos/Opengl/lucesTextOpenGL/HTM/texturas.html>. [Último acceso Octubre 2011].

[70] "Android Programming 3D Graphics with OpenGL ES" . Disponible en World Wide Web: <http://www3.ntu.edu.sg/home/ehchua/programming/android/Android_3D.html>. [Último acceso Octubre 2011].

[71] Ed Burnette, "Hello, Android". Disponible en World Wide Web: <http://kronox.org/documentacion/Hello.Android.new.pdf>. [Último acceso Octubre 2011].

[72] Mercedes Sánchez Marcos y Almudena Sardón García, Universidad de Salamanca, "Luces y lámparas". Disponible en World Wide Web: < http://gsii.usal.es/~igrafica/descargas/temas/Tema08.pdf>. [Último acceso Octubre 2011]